Rotor Module for the Structural Dynamics Toolbox

For Use with MATLAB®

VERY EXPERIMENTAL.
NOT ALL FUNCTIONALITY IS DOCUMENTED
Some functionality in the manual
may not be distributed

User's Guide

Version 1.0

Etienne Balmes

Jean-Philippe Bianchi

Arnaud Sternchüss

How to Contact SDTools

33 +1 44 24 63 71	Phone
33 +6 77 17 29 99	Fax
SDTools 44 rue Vergniaud 75013 Paris (France)	Mail

www.sdtools.com	Web
support.sdtools.com	Support web site
support.sdtools.com/gf/project/openfem/	The FEM engine developped by SDT
info@sdtools.com	Sales, pricing, and general information

SDT Rotor Module User's Guide on October 1, 2012

© Copyright 1991-2012 by SDTools

The software described in this document is furnished under a license agreement.
The software may be used or copied only under the terms of the license agreement.
No part of this manual in its paper, PDF and HTML versions may be copied, printed, photocopied or reproduced in any form without prior written consent from SDTools.

Structural Dynamics Toolbox is a registered trademark of SDTools
OpenFEM is a registered trademark of INRIA and SDTools
MATLAB is a registered trademark of The MathWorks, Inc.
Other products or brand names are trademarks or registered trademarks of their respective holders.

Contents

1 Installation **5**

2 Theoretical reminders **7**
2.1 Rotating bodies 8
2.1.1 Problem definition in a rotating frame 8
2.2 Problem definition in a fixed frame 10
2.3 Fourier analysis of structures 10
2.3.1 cyclic structure basics 11
2.3.2 Fourier transform for shaft computations 12
2.3.3 Solutions in periodic media 12

3 Toolbox tutorial **15**
3.1 Rotor meshing 17
3.1.1 Meshing utilities 17
3.1.2 Basic 1D rotor example 19
3.1.3 Meshing 3D rotor from 1D and 2D models 20
3.1.4 From sector to shaft in the case of cyclic symmetry 22
3.1.5 Utilities for handling slanted blades 23
3.1.6 Disk connections in multi-stage cyclic symmetry 23
3.1.7 View meshes for cyclic symmetry 25
3.2 Bearing and support representations 26
3.2.1 Linear bearing 27
3.2.2 Non-linear bearings in the time domain 28
3.3 Gyroscopic effects 28
3.3.1 Fixed frame models 28
3.3.2 Rotating frame models 29
3.4 Frequency domain analysis, full model 29
3.4.1 Campbell diagrams, full model 30
3.4.2 Blade with centrifugal stiffening 30

3.4.3 Complex modes . . . 31
3.4.4 Forced frequency response to unbalanced load . . . 32
3.5 Solvers for models with cyclic symmetry . . . 33
3.5.1 Static response . . . 33
3.5.2 Single stage mode computations . . . 34
3.5.3 Multi-stage harmonic mode computations . . . 35
3.5.4 Campbell diagrams . . . 36
3.5.5 Complex modes . . . 37
3.5.6 Forced frequency response to unbalanced load . . . 37
3.6 Full rotor model from cyclic computation . . . 37
3.6.1 Single stage full rotor example . . . 37
3.7 Time domain analysis . . . 38
3.7.1 Simple example . . . 39
3.7.2 Gyroscopic effects . . . 39
3.7.3 Other representations of bearings . . . 42

4 Validation 43
4.1 Rigid disk example . . . 44
4.1.1 Matrices in rotating frame . . . 44
4.1.2 Matrices in global fixed frame . . . 45
4.1.3 Validation with 3D model disk . . . 47
4.2 Simple 2DOF model of shaft with disk . . . 50
4.3 1D models . . . 54
4.3.1 1D example in a fixed frame . . . 54
4.3.2 1D models in a rotating (body-fixed) frame . . . 57
4.4 3D rotor . . . 57
4.5 Data structure reference . . . 59

5 Function reference 61
fe_cyclic . . . 62
fe_rotor . . . 65
rotor1d . . . 71
rotor2d . . . 74
demo_cyclic . . . 75
fe_cyclicb Mesh . . . 79
fe_cyclicb . . . 83
obsolete . . . 95
nl_spring . . . 103
mkl_utils . . . 112
Non linearities list . . . 114

Creating a new non linearity: nl_fun.m 130
nl_solve 132
nl_mesh 135
spfmex_utils 142
extrotor 144

Bibliography **145**

Index **146**

Installation

Contents

The SDT/Rotor toolbox is intalled as a patch on an SDT installation. You can download the toolbox from

`http://www.sdtools.com/distrib/beta/rotor_patch_dis.p`

Then within Matlab,

```
% 1/ cd to location where you saved the patch
%  MUST differ from SDT path
cd tempdir
% 2/ check that you will overwrite the expected SDT installation
path
which('feplot');
% 3/ install the patch (you must have write permission on the directory)
rotor_patch_dis
```

2

Theoretical reminders

Contents

2.1 Rotating bodies 8
2.1.1 Problem definition in a rotating frame 8
2.2 Problem definition in a fixed frame 10
2.3 Fourier analysis of structures 10
2.3.1 cyclic structure basics 11
2.3.2 Fourier transform for shaft computations 12
2.3.3 Solutions in periodic media 12

The following sections give a number of theoretical reminders on things used for the toolbox. THIS IS VERY INCOMPLETE AND NOT VERY ORDERED.

2.1 Rotating bodies

2.1.1 Problem definition in a rotating frame

The developments of this section are derived from internal work on the SDT Rotor module which is currently only distributed to SNECMA. The results shown here can be seen as a summary of those found in Ref. [1] which treats the problem with a strong emphasis on the theoretical formalism. Other classical references that treat of the problem of rotating bodies are [2],[3], [4].

Particles located in point p of the body fixed frame are at location x at time t. One defines the displacement u by

$$\{x(p,t)\} = \{p\} + \{u(p,t)\} \tag{2.1}$$

At time t, a reference point of the rotating body is assumed to have a rigid rotation speed $\{\omega\}$ with respect to the reference frame (in the present study, this speed is related to a global rotation around a fixed axis characterized by angle θ). The velocity is thus given by

$$\{v_x\}(p,t) = \frac{\partial\{x\}}{\partial t} + \{\omega(t)\} \wedge \{x(p,t)\} \tag{2.2}$$

This expression can easily be derived by decomposing the position in body fixed coordinates $\{x\} = x_i \{e_{bi}\}$ and noting that the derivatives the base vectors $\partial\{e_{bi}\}/\partial t = \{\Omega\}(t) \wedge \{e_{bi}\}$. In implementations, one replaces the vector product $\omega(t)\wedge$ by the product by the corresponding skew-symmetric matrix

$$\Omega(t) = \begin{bmatrix} 0 & -\omega_z(t) & \omega_y(t) \\ \omega_z(t) & 0 & -\omega_x(t) \\ -\omega_y(t) & \omega_x(t) & 0 \end{bmatrix}. \tag{2.3}$$

The acceleration, derived from the velocity expression, is given by

$$\begin{array}{rcl} \{a\}(p,t) & = & \{\ddot{x}\} \\ & + & \left[\dot{\Omega}(t)\right]\{x\} + 2\left[\Omega\right]\{\dot{x}\} \\ & + & \left[\Omega^2\right]\{x\} \\ & = & \{\ddot{u}\} + \left[\dot{\Omega}(t)\right](p+u) + 2\left[\Omega\right]\{\dot{u}\} + \left[\Omega^2\right]\{p+u\} \end{array} \tag{2.4}$$

where three contributions (rows of the equation) are typically considered : the acceleration in the rotating frame, the Coriolis acceleration and the centrifugal acceleration.

The virtual work of acceleration quantities is thus typically expressed as

$$\{\hat{q}\}\left([M]\{\ddot{q}\}+[D_g]\{\dot{q}\}+[K_a+K_c]\{q\}+f_c+f_g\right) \tag{2.5}$$

with the following element level expressions. The displacement within an element is given by the position and the element shape functions $[N]$ in the three directions xyz

$$\{x\}=\{p\}+[N_{xyz}]\{q\}$$

The matrices and loads are integrated over the volume $\mathcal{S}_0$ in the reference configuration and are given by

- $[M]=\int_{\mathcal{S}_0}\rho_0\left[N_{xyz}\right]^\top[I]\left[N_{xyz}\right]\ d\mathcal{S}_0$ the mass in the rotating frame
- $[D_g]=\int_{\mathcal{S}_0}2\rho_0\left[N_{xyz}\right]^\top[\Omega]\left[N_{xyz}\right]\ d\mathcal{S}_0$ the gyroscopic coupling
- $[K_c]=\int_{\mathcal{S}_0}\rho_0\left[N_{xyz}\right]^\top\left[\Omega^2\right]\left[N_{xyz}\right]\ d\mathcal{S}_0$ the centrifugal softening/stiffening
- $[K_a]=\int_{\mathcal{S}_0}\rho_0\left[N_{xyz}\right]^\top\frac{\partial[\Omega]}{\partial t}\left[N_{xyz}\right]\ d\mathcal{S}_0$ the centrifugal acceleration
- $f_c=\int_{\mathcal{S}_0}\rho_0\left[N_{xyz}\right]^\top\left[\Omega^2\right]\{p\}\ d\mathcal{S}_0$ the centrifugal load
- $f_g=\int_{\mathcal{S}_0}\rho_0\left[N_{xyz}\right]^\top\frac{\partial[\Omega]}{\partial t}\{p\}\ d\mathcal{S}_0$ the Coriolis load

It is acknowledged that the notations used can be somewhat confusing. Indeed, in a discretized vectors DOFs are placed either sequentially x,y,z at all nodes of the element, or separated x at all nodes, ... while the operations $\left[N_{xyz}\right]^\top[I]\left[N_{xyz}\right]$ imply the use of vectors. This 2 dimensional product notation however directly reflects the numerical implementation as is thus deemed preferable.

In the applications considered in this study, one will use a fixed axis of rotation $\Omega=\omega(t)\{e_z\}\wedge$. The matrices and loads are thus proportional to the scalars ω, ω^2 and $\dot{\omega}$. One will thus simply use

$$[D_g(\omega)]=\omega\,[D_g(1)]$$

which results in significant computational cost savings since the matrix only needs to be computed for a single velocity. One proceeds similarly for the other matrices and loads.

2.2 Problem definition in a fixed frame

Fully axisymmetric rotors can be modeled in a fixed frame using an Eulerian representation, where particles are moving under a deformed mesh. Particles located at point $\{p(p_0,t)\} = \{p_0\} + \{u(p_0,t)\}$ in the deformed Eulerian frame have a velocity given by xxx

$$\{v(x,t)\} = \frac{\partial x(p+u(p,t),t)}{\partial t} + \frac{\partial x(p+u(p,t),t)}{\partial x} \tag{2.6}$$

The body fixed frame verifies $\theta = \phi + \Omega t$, which can be written as $p_G = \{r,\theta,z\} = R_{\Omega t}\{p\}$. The matching of displacements in both frames is given by

$$\{u(p,t)\} = \{u_G(p_G,t)\} \tag{2.7}$$

The velocity of a particle in the disk is given by

$$\{v(p_G,t)\} = [\Omega]\{p_G\} + [\Omega]\left\{\frac{\partial u}{\partial p_G}\right\} + \left\{\frac{\partial u_G}{\partial t}\right\} \tag{2.8}$$

Validation example : One first considers a disk that has a steady state deformation in the global frame. That is the Eulerian frame, one has $\frac{\partial u}{\partial t} = 0$ and $\{u(r,\theta,z)\} = \{u(p)\} = (2 - cos^2\theta)\{e_\theta\}$.

One now considers a disk that has a steady state deformation in the rotating frame: $\frac{\partial u}{\partial t} = 0$ and $\{u(r,\phi,z)\} = (1 - cos^2\phi)\{e_\phi\}$. The displacement of a particle located at at time t is given by

$$\{u_G(p_G,t)\} = (1 - cos^2(\theta - \Omega t))\left\{e_{(\theta-\Omega t)}\right\} = [R_{\Omega t}]\{u(p)\}$$

its velocity is

2.3 Fourier analysis of structures

For more details, you can refer to[5] that is available on-line.

2.3.1 cyclic structure basics

For cyclic system with N sectors (angle $\alpha = 2\pi/N$), a point load placed at angle $n\alpha$ is associated with the harmonic load

$$Y(k) = \frac{1}{N}\sum_{n=0}^{N-1} y(n)e^{-j\alpha nk} \tag{2.9}$$

this can be used to simply compute the k diameter response. Using the symmetry of the spectrum, one can recover the full spatial response by inverse fourier transform. For N odd

$$y(n) = Y(0) + \sum_{k=1}^{(N-1)/2} 2\mathrm{Re}(Y(k)e^{j\alpha nk}) \tag{2.10}$$

for N even

$$y(n) = Y(0) + (-1)^n Y(N/2) + \sum_{k=1}^{N/2-1} 2\mathrm{Re}(Y(k)e^{j\alpha nk}) \tag{2.11}$$

The displacements $y(n)$ of each sector are however expressed in local coordinates, when applying conditions a transformation $\theta^{\alpha n}$ to global coordinates is thus needed

$$\left\{ \begin{array}{c} q(n)_x \\ q(n)_y \\ q(n)_z \end{array} \right\}_{Glob} = \left[\begin{array}{ccc} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{array} \right]^n \left\{ \begin{array}{c} q(n)_x \\ q(n)_y \\ q(n)_z \end{array} \right\}_{Loc} \tag{2.12}$$

For a point on the axis, the in plane response is given by two components $u(0), v(0)$, the response at other points of the series is given by (2.10) but the point being coincident on has $\{u(0), v(0)\} = (\theta^{\alpha})^n \{u(n), v(n)\}$. It follows that $Y(k) = 0$ for all $k \neq 1$.

The intersector continuity condition says that the nodes that are common to two sectors have the same motion. By convention the left nodes in SDT rotor are those with the lowest clockwise polar angle, the left nodes of sector 0 have thus equal motion than the right nodes or sector $N-1$. Thus $c_L - c_R(Y(k)e^{-j\alpha k}) = 0$. Which leads to the constraint

$$\left[\begin{array}{cc} [c_l] - \cos(k\,\alpha)\,[c_r] & -\sin(k\,\alpha)\,[c_r] \\ \sin(\delta\,\alpha)\,[c_r] & [c_l] - \cos(\delta\,\alpha)\,[c_r] \end{array} \right]_{2N_r \times 2N} \left\{ \begin{array}{c} Re(q) \\ Im(q) \end{array} \right\} = 0 \tag{2.13}$$

The Fourier transform being a linear relation, one can actually rewrite the relation as a $\{y\} = [E]\{Y\}$. For N even

$$\begin{Bmatrix} y_0 \\ \vdots \\ y_n \\ \vdots \\ y_{N-1} \end{Bmatrix} = \begin{bmatrix} 1 & \cdots & 2 & 0 & \cdots & 1 \\ \vdots & & \vdots & \vdots & & \vdots \\ 1 & \cdots & 2cos(\alpha kn) & -2sin(\alpha kn) & \cdots & (-1)^n \\ \vdots & & \vdots & \vdots & & \vdots \end{bmatrix} \begin{Bmatrix} Y_0 \\ \vdots \\ Re(Y_k) \\ Im(Y_k) \\ \vdots \\ Y_{N/2} \end{Bmatrix} \tag{2.14}$$

Note that $E^T E$ is a diagonal matrix with $[N\ 2N\ \cdots\ 2N\ N]$ on the diagonal.

Posing $[I_{NQ}]$ is the identity matrix whose size is the number of physical DOF of the sector and $\otimes$ is the Kronecker product, the fourier DOFs Q_k (sorted as in (2.14)) and physical DOFs on the whole disk are related by

$$\{q\} = [E \otimes I_{NQ}]\{Q_k\} \tag{2.15}$$

When performing harmonic computations, one typically uses the fact that the model matrices are identical for each sector. As a result, the transformed matrices are block-diagonal, which is the basis for the ability to compute full responses based on independent computation of each Fourier solution Q_k.

When adding a matrix to sector n, the relation to to the Fourier DOFs is given by a row E_n, the product $E_n^T E_n$ is not block diagonal as a result one has coupling between the Fourier harmonics.

2.3.2 Fourier transform for shaft computations

To simplify computations XXX

2.3.3 Solutions in periodic media

This really does not apply to rotors but is implemented in `fe_cyclic`, and thus documented here. One considers a model whose properties are spatially periodic. For a physical response, known at regularly spaced positions $n\Delta x$, one can compute the its fourier transform

$$U(\kappa) = \sum_{n=-\infty}^{+\infty} u(n\Delta x) e^{-jn(\kappa \Delta x)} \tag{2.16}$$

$U(\kappa)$ is a complex shape defined on the mesh of the repeated cell. One actually uses two cells to represent the real and imaginary parts of U. The wave number κ varies in the $[0, \frac{2\pi}{\Delta x}]$ interval (or any interval of the same length, since $U(\kappa)$ is periodic in the wavelength domain).

Given the fourier transform $U(\kappa)$, on can recover the physical motion by computing the inverse Fourier transform

$$u(n\Delta x) = \frac{\Delta x}{2\pi} \int_0^{\frac{2\pi}{\delta x}} U(\kappa) e^{j\kappa n \Delta x} d\kappa \tag{2.17}$$

For a mono-harmonic response (fixed wave number κ), the spatial transform is given by $u(n\Delta x) = \left(U(\kappa)e^{jn(\kappa\Delta x)}\right)$, using the continuity condition linked to the fact that $u_{left}(n\Delta x) = u_{right}((n-1)\Delta x)$, one thus has $[c_L]\{U(\kappa)\} = [c_R]\{U(\kappa)\}e^{-j(\kappa\Delta x)}$. Hence in the real/imaginary format, one has the constraint equation

$$\left[\begin{array}{cc} [c_l] - \cos(\kappa\Delta x)[c_r] & -\sin(\kappa\Delta x)[c_r] \\ \sin(\kappa\Delta x)[c_r] & [c_l] - \cos(\kappa\Delta x)[c_r] \end{array}\right]_{2N_r \times 2N} \left\{\begin{array}{c} Re(U) \\ Im(U) \end{array}\right\} = 0 \tag{2.18}$$

and one solves for $U(\kappa)$ knowning the transform of the applied loads. For a discrete load on the first slice $n = 0$, the fourier coefficients of the load is $F(\kappa) = f(0)$.

One can usefully note that the wave length $L = \pi/\kappa$ covers the full interval of positive lengths, that $U(\frac{2\pi}{\delta x} - \kappa) = \bar{U}(\kappa)$, and that the half spectrum corresponds to a wavelength of Δx. Using the property of symmetry allows the use of computations for wavelengths larger than Δx only. In the `fe_cyclic eig` calls, one specifies the period as a number of steps $p\Delta x$. The phase angle is thus $\kappa\Delta x = \frac{2\pi}{p}$ and the spectrum is symmetric for $p' = 2/p$.

```
mo1=femesh('testhexa8b divide 2 1 1');
mo1=fe_cyclic('build -1 1 0 0',mo1);

% symmetry along right edge
r2=fe_cyclic('build -1 0 1 0',mo1);
r2=fe_case(r2,'getdata','Symmetry');
mo1=fe_case(mo1,'FixDof','y_symmetry',r2.IntNodes(:,1)+.02);

mo1=stack_set(mo1,'info','EigOpt',[2 2 1e3]);
range=1./linspace(.01,.99,21);
def=fe_cyclic(['teig -all' sprintf(' %g',range)],mo1);
figure(1);plot(1./def.data(:,2),def.data(:,1),'x')
```

```
feplot(mo1,def)

[r1,i1]=unique(def.data(:,2),'first');
Ek=fe_caseg('enerm -bygroup',mo1,fe_def('subdef',def,i1));
if norm(Ek.Y*4-1)>sqrt(eps); error('Inconsistent energy');end
if 1==2 % manual verification
 [m,k,mdof]=fe_mknl(mo1,'assemble NoT');
 def=feutilb('placeindof',mdof,def);
 feutilb('dtkt',real(def.def),m)+feutilb('dtkt',imag(def.def),m)
end
```

3

Toolbox tutorial

Contents

3.1 Rotor meshing 17
3.1.1 Meshing utilities 17
3.1.2 Basic 1D rotor example 19
3.1.3 Meshing 3D rotor from 1D and 2D models 20
3.1.4 From sector to shaft in the case of cyclic symmetry 22
3.1.5 Utilities for handling slanted blades 23
3.1.6 Disk connections in multi-stage cyclic symmetry 23
3.1.7 View meshes for cyclic symmetry 25
3.2 Bearing and support representations 26
3.2.1 Linear bearing 27
3.2.2 Non-linear bearings in the time domain 28
3.3 Gyroscopic effects 28
3.3.1 Fixed frame models 28
3.3.2 Rotating frame models 29
3.4 Frequency domain analysis, full model 29
3.4.1 Campbell diagrams, full model 30
3.4.2 Blade with centrifugal stiffening 30
3.4.3 Complex modes 31
3.4.4 Forced frequency response to unbalanced load 32
3.5 Solvers for models with cyclic symmetry 33
3.5.1 Static response 33
3.5.2 Single stage mode computations 34
3.5.3 Multi-stage harmonic mode computations 35
3.5.4 Campbell diagrams 36
3.5.5 Complex modes 37
3.5.6 Forced frequency response to unbalanced load 37
3.6 Full rotor model from cyclic computation 37

3.6.1 Single stage full rotor example 37
3.7 Time domain analysis 38
3.7.1 Simple example 39
3.7.2 Gyroscopic effects 39
3.7.3 Other representations of bearings 42

fe_rotor module can use 2 different frames to describe rotating effects (rotating frame and fixed frame). Rotors can also be describes at 3 levels of modelization : 1d, 2d or 3d.

The next section illustrates meshing capabilities, supported computations are described next

- Frequency domain analyses (section 3.4) : Campbell diagram building, Direct computation of critical speeds, response to unbalanced mass, asynchronous load, and harmonic loads on bearings.
- Time domain simulation accounting for non-linear bearings is under development (these are performed with the shaft in a rotating frame and the stator fixed).

3.1 Rotor meshing

3.1.1 Meshing utilities

1D

The SDT/Rotor toolbox supports analysis of 1D models of symmetric rotors composed of

- shafts represented by beam1 element (see sdtweb('beam1') and sdtweb('p_beam') The rotation axis is taken to be that of the beam.
- disks represented by mass1 (see sdtweb('mass1')). The rotation axis is the one whose moment of inertia is different from the 2 others that are equal.
- bearings supports by celas.

You can generate a beam model of your rotor by providing a skyline (points not on the axis defining the radius at various locations). Use NaN to define 2 segments. See rotor1d Skyline for more details.

```
xy=[0 0; 0 .1;1 .1;1 1;1.1 1;1.1 .1;2 .1];
xy(:,3)=0;  figure(1);plot(xy(:,1),xy(:,2));
% Mesh as beams
mo1d=rotor1d('skylineToBeam',xy);
% Add bearings as spring elements:
mo1d=rotor1d('AddBearing DOF -123 k 1e4 -keep',mo1d,[0.1 0 0]);
mo1d=rotor1d('AddBearing DOF -123 k 1e4 -keep',mo1d,[1.9 0 0]);

% now view as 3D model
mo2d=rotor1d('1To2d lc 5e-2',mo1d);
cf=feplot;cf.model=rotor2d('buildFrom2D nsec16',mo2d);
```

You can also add beam through `rotor1d Addbeam` command.

```
mo1d=rotor1d('AddBeam x1 0 x2 0.7 r2 0.77 MatID 1',[]); % add rod
mo1d=rotor1d('AddBeam x1 0.5 x2 0.8 r1 0.77 r2 0.90 MatID 1',mo1d); %add tube
mo2d=rotor1d('1To2d lc 5e-2',mo1d);
cf=feplot; cf.model=mo2d; fecom colordatapro;
```

2D

For 2D rotor representations, the `SkyLineTo2d` command eases the generation of simple rotors. Note how in the following example `Nan` separators are used to generate a rotor in multiple parts : center shaft first, then after the separator various disks.

```
xy=[ 0 0;0  0.00945;0.0088  0.00945; 0.0088  0.0057;
0.042  0.0057;0.042  0.00938; 0.057  0.00938;
0.057  0.008;0.0637  0.008; 0.0637  0.00595;
0.0919  0.00595;0.0919  0.00925; 0.0979  0.00925;
0.0979  0.006;0.121  0.006; 0.121  0.008;
0.127  0.008;0.127  0.0095; 0.142  0.0095;
0.142  0.006;0.175  0.00565; 0.226  0.00565; 0.226  0.0088;
0.232  0.0088;
NaN 0.0057; % Disk
0.00952  0.0057;0.00952  0.008;
 0.0209  0.008; 0.0209  0.0369;
 0.0242  0.0369; 0.0242  0.0572; 0.0266  0.0572;0.0266  0.0369;
 0.0299  0.0369; 0.0299  0.008;0.0413 0.008
NaN .00565; % Separator giving non zero internal diameter .00565
0.177+.01  0.0057;0.177+.01  0.02;0.191  0.02; 0.191  0.0057;
NaN .00565; % Separator giving non zero internal diameter .00565
```

```
0.197+.005  0.0374; 0.215-.005  0.0374;  % Impeller
 0.215-.005  0.006+.003;0.222-.002  0.006+.003; % between sub disks
 0.222-.002  0.0775; 0.2259  0.0775;
];
xy(:,3)=0;
mo2d=rotor1d('skyline To2d -lc .005',xy); % x axis
mo2d.pl=[ ...
   1 fe_mat('m_elastic',1,1)  200000000000 0.29 7800
   2 fe_mat('m_elastic',1,1)  7.17e10 0.33 2830 %2830
   3 fe_mat('m_elastic',1,1)  3.07e12 0.3 .78 % 7800
];
mo2d.Elt(feutil('findelt matid 2 3 4 5',mo2d),5:6)=2;
mo2d.Elt(feutil('findelt innode {x>=.15 & y<=.0057}',mo2d),5:6)=3;
mo2d.Elt(feutil('findelt innode {x>=.22 & y<=.01}',mo2d),5:6)=3;
feplot(mo2d); fecom colordatamat

cf=feplot;cf.model=rotor2d('buildFrom2D nsec16',mo2d);
cf.sel={'-disk1','ColorDataMat'};fecom(';view1')
```

Figure 3.1: Example rotor generated with meshing utilities

3.1.2 Basic 1D rotor example

Following example builds by hand a simple 1D rotor, with one shaft, one disk and 2 bearing stifnesses. It is almost the same as one accessible through `d_rotor TestShaftDiskMdl`.

```
 % define mesh :
 model=struct('Node', ...
         [1 0 0 0  0 0 0; 2 0 0 0  0.4/3 0 0;
```

```
          3 0 0 0  0.4*2/3 0 0; 4 0 0 0  0.4 0 0]);
model.Elt=feutil('ObjectBeamLine',(1:4)'); % define shaft
model.Elt=feutil('ObjectMass',model,2,...
    [16.5 16.5 16.5 0.18608 0.093 0.093]); % add disk
model=feutil('AddElt',model,'celas', ...
     [3 0 2 0 100 0 0; 3 0 3 0 101 0 0]); % add bearings (y and z stifness)
% define properties
model.pl=m_elastic('dbval 1 steel');  % shaft material
model.il=p_beam('dbval 1 circle 1e-2'); % shaft, r=1e-2
model.il=p_spring(model.il,'dbval 100 5e5','dbval 101 5e5'); % bearings
% ends boundaries :
model=fe_case(model,'fixdof','Ends',...
    [1.01;1.02;1.03;4.01;4.02;4.03]);
% Assemble nominal matrices:
model=fe_caseg('assemble -reset -secdof -matdes 2 1 70',model);
cf=feplot(model);
% For solution see sdtweb('freqstud')
```

Note that at this time only fixed frame representation is available for such 1d rotors (`beam1` and `mass1` elements). Gyroscopic coupling is then computed under `MatType 70` (The formula for gyroscopic coupling can be found in [6]) The nominal representation for these models is then the Eulerian point of view where the displacement of the rotor is expressed in a non-rotating frame. For the 1D representation, the model nodes are always placed on the nominal rotation axis. Thermal and pre-stress effects are not accounted for.

3.1.3 Meshing 3D rotor from 1D and 2D models

The rotor module supports all 3D elements of SDT. 2D models are only considered through an extrusion and 3D cyclic symmetry. One can import a volume model, or mesh it using `feutil` meshing commands. This section describes procedures to mesh volumes from 1D and 2D models.

From 1D rotor model meshed using `beam1` elements, one can create a 2D model using `rotor1d 1To3D` command. Note that `mass1` elements can not be converted for the moment. For example, with section **??** rotor model

```
xy=[0 0; 0 .1;1 .1;1 1;1.1 1;1.1 .1;2 .1];
mo1d=rotor1d('skyline ToBeam',xy); % Mesh as beams
% Add bearings as spring elements:
```

```
mo1d=rotor1d('AddBearing DOF -123 k 1e4 -keep',mo1d,[0.1 0 0]);
mo1d=rotor1d('AddBearing DOF -123 k 1e4 -keep',mo1d,[1.9 0 0]);
mdl3d=rotor1d('1To3d-quad-lc0.02-div24',mo1d); % build 3d rotor
cf=feplot(mdl3d)
```

Command option `-quad` force the use of quadratic elements (`hexa20`) instead of linear elements (`hexa8`).
Shaft is meshed using `hexa8` degenerated elements. The edges on the axis of the shaft are using the same nodes as the beam nodes so that bearings described by celas in 1D intial rotor can remain the same. RBE3 rings are created at each bearing celas. Concentrated masses (no inertia) on the axis are also left.
THIS IS NO LONGER TRUE BUT SHOULD BE REACTIVATED XXX: `mass1` elements with inertia that represents disks, are meshed as volume disks of arbitrary thickness (dt) 0.005, with radius R_2 computed so that inertia along rotation axis is the same ($Ir = 0.5m(R_2^2 + R_1^2)$) and then density is computed to match the mass m ($m = \pi * (R_2^2 - R_1^2) * dt$). ($R_1$ is shaft radius). Young modulus of disk is taken at 100*steel modulus.

Gyroscopic coupling and centrifugal stifness for 3D elements are only described in the rotating frame, that is to say under `MatType 7` and `MatType 8`.

```
mdl3d=fe_mat('defaultil',mdl3d); % default element pro
mdl3d=fe_mat('defaultpl',mdl3d); % default material pro
% Assemble nominal matrices:
mdl3d=fe_caseg('assemble -reset -secdof -matdes 2 1 7 8',mdl3d);
```

One can also find an example of a 3D rotor in `d_rotor('TestVolShaftDiskMdl')`.

```
data={'Rs'  ,0.01  , 'shaft radius';
      'Rd'  ,0.15  , 'disk radius';
      'Ls'  ,0.4   , 'shaft length';
      'Ld'  ,0.03,   'disk thickness';
      'yd'  ,0.4/3,  'disk position on the shaft'
      'yb'  ,0.4*2/3,'bearing stifness position on the shaft'
      'Tol' ,0.05,   'elt length Tol'
      };
model=d_rotor('TestVolShaftDiskMdl',data);
```

`rotor2d BuildFrom2D` convert 2d model to 3d model using cyclic symetry. For example, to convert previous 2D model:

```
mdl2d=rotor1d('1To2d-quad-lc0.02',mo1d);
mdl3d=rotor2d('buildFrom2D -close nsec3 div8',mdl2d);
```

3.1.4 From sector to shaft in the case of cyclic symmetry

Closing a disk

The SDT `fe_cyclic` function only handles single sector models using symmetry conditions. SDT/Rotor extends the capabilities by dealing with a full (possibly multi-stage, then called shaft) rotor model.

Building a disk/shaft model is done in two steps. For each sector, one defines matching edges using a `fe_cyclic Build` command, the generates a disk/shaft model using `fe_cyclicb DiskFromSector`.

```
demosdt(['download-back ' ...
 'http://support.sdtools.com/gf/download/forummessage/4/11/donnees_secteur.dat'
% -removeface : removes skin elements
samcef('read-removeface',fullfile(sdtdef('tempdir'),'donnees_secteur.dat'));
cf=feplot;
% Fix sector edges
fe_cyclic('buildepsl .1 fix',cf.mdl);
fecom('curtabCases','Symmetry')
fecom(';view3;proViewOn');
```

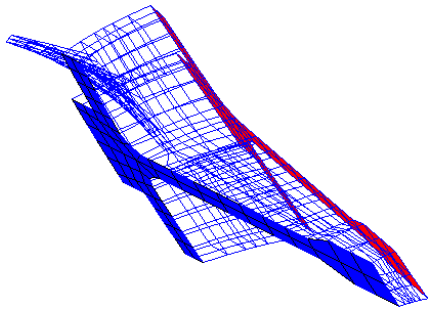

Figure 3.2: Left and right sides of the sector

The left and right edges of sectors should be conform ! Practically you need to mesh the two surfaces first (possibly generate the second by rotation of the first). Then mesh the interior. You will note that this typically requires at least two layers of element for tetra10 meshing.

3.1.5 Utilities for handling slanted blades

Typically blades have an angle with respect to the e_r, e_z plane. In a number of cases, removing this angle makes node and element selection easier. With two nodes the FixTheta command modifies node positions by removing the angle. The transformation is saved in info,FixTheta stack entry and the back transformation is obtained using a FixTheta call with no argument.

```
cf=feplot(2);cf=d_rotor('TestSector');%sdtweb('d_rotor.m#TestSector')
n1=cf.mdl.Node; % save nodes

cg=feplot(5); % place rectified model in figure(5)
cg.mdl=fe_cyclicb('MeshFixTheta 10061 10086 -offset 80',cf.mdl);
fecom(cg,'view2')
% verify that the back step works
cf.mdl=fe_cyclicb('MeshFixTheta',cg.mdl);fecom(cf,'view1')
if norm(cf.mdl.Node-n1)>1e-10; error('Back transform failed');end
```

3.1.6 Disk connections in multi-stage cyclic symmetry

In multi-stage cyclic symmetry, each stage is modeled using a superelement called disk*i* (see fe_cyclic DiskFromSector). Coupling between stages is done using elements. The most consistent approach is to use a physical area that is properly meshed for the full 360 degrees, but this may be difficult in particular when the mesh refinement is notably different between the two stages, so that a node to surface penalized connection is also implemented and an example given at the end of this section.

Two steps are required:

1. the first step is a manual declaration of the nodes that belong to the two regarding surfaces. The declaration of the nodes in the 2D cut provided by fe_cyclicb DisplayAllEdges command is sufficient (and therefore recommended).

2. the second step is the automatic reconstruction of the rims as volumes (using automated 3D tessellation) or **penalty** based node to surface bilateral contact in the MeshRim command.

   Note that **fe_shapeoptim** can be used to local deform the disk in order to allow rim meshing.

```
% load two disk example with space between disks
cf=demo_cyclic('TestForMeshRimVol')

% RimStep1: select nodes at the matching interfaces
fe_cyclicb('DisplayAllEdges',cf);
% start cursor to pick values : fe_fmesh('3dlineinit')
n1=[12 18 24 1127 1133 1139];

% RimStep2: build rims and tessellate
fe_cyclicb('MeshRimStep2 epsl.1',cf,n1);
cf.sel='EltName~=SE';fecom('showpatch');
```

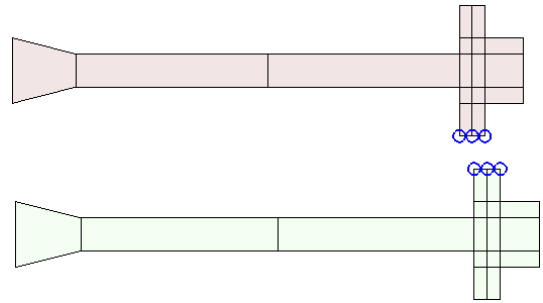

Figure 3.3: Selected disk edge nodes for rim tesselation

If you choose the **penalty** approach here is a working example where the edges of sector edges are assumed coincident thus allowing an automated search of intersection with the `-FindIntersect` option.

```
% load two disk example with space between disks
cf=demo_cyclic('testrotor 7 10 3 -NoRim -RimH .1 -blade -cf 2 reset');

% RimStep1: find nodes at the matching interfaces
n1=fe_cyclicb('DisplayAllEdges -FindIntersect epsl .2',cf);

% RimStep2: build rims as springs
fe_cyclicb('MeshRimStep2 epsl.1 -kp 1e12 -slavedisk 1 3 -masterdisk 2',cf,n1);

cf.sel={'groupall','colordatagroup -edgealpha .05 -alpha.1'}
def=fe_cyclicb('shaft eig 0',cf.mdl);
sel={'disk1','groupall';'disk2','groupall';'disk3','groupall';'rim',''};
```

```
 cf.def=fe_cyclicb('DisplaySel',cf,def,sel);
fecom('ColorDataEvalTanz')
```

3.1.7 View meshes for cyclic symmetry

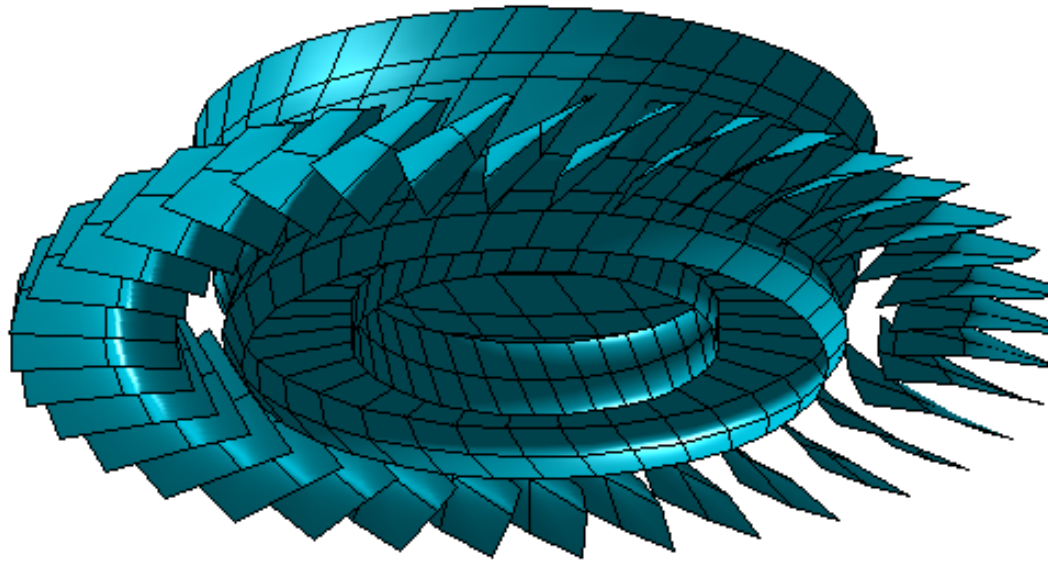

Figure 3.4: Sample viewing mesh for post-processing multi-stage computations

Meshing tools also include procedures to build viewing meshes from the finite element mesh. `fe_cyclicb('MeshRimLine2Patch',cf,sel)` aims to build viewing meshes made of surface elements connecting selected nodes of the true mesh of the rotor. `sel` can be:

- an array of lines connecting nodes of the inner disk parts in the 2D cut of the rotor (returned by a previous `fe_fmesh('3dLineInit')`). Note that this function automatically removes the segments that connect nodes of two different disks.
- an array of elements connecting nodes of the blades of each disk (returned by a previous `fe_fmesh('3dLineInitAddInfo Quad4')`).

The selection is stored in the `cf.mdl.Stack{'info','ViewMesh'}`. If the `-reset`

option is not specified, the current selection is appended to the existing one. For each project, you should typically edit a script similar to the following

```
cf=demo_cyclic('BuildStep0');

% Viewing mesh step 1: disk elements
fe_cyclicb('DisplayAllEdges',cf);
%fe_fmesh('3dLineInit',cf); % right click 'Type' or 'Done'
L=[1 3 5 15 26 0 1121 1123 1125 1135 1146 0 13 15 18 1133 1135 1138];
fe_cyclicb('MeshRimLine2Patch -reset',cf,L);

% Viewing mesh step 2: blade elements
% disk1
fe_cyclicb('DisplayFirst',cf,{'disk1'});
%fe_fmesh('3dLineInitAddInfo Quad4',cf); % pick four nodes to form a quad
% use right click 'Type' or 'Done' to display
Elt=[Inf abs('quad4');
154 156 152 148 1 1;148 146 150 154 1 1;
146 142 144 150 1 1;142 111 83 144 1 1];
fe_cyclicb('MeshRimLine2Patch',cf,Elt);

% disk2
fe_cyclicb('DisplayFirst',cf,{'disk2'});
fe_fmesh('3dLineInitAddInfo Quad4',cf); % pick four nodes to form a quad
model.Elt=[Inf abs('quad4');
1274 1276 1272 1270 1 1;1270 1272 1268 1266 1 1
1266 1268 1264 1262 1 1;1262 1231 1203 1264 1 1];
fe_cyclicb('MeshRimLine2Patch',cf,model.Elt);
fecom('ShowPatch');
% save cf.Stack{'info','ViewMesh'}

% Once the geometry generated typical calls are
fesuper('Sebuildsel -initrot',cf,cf.Stack{'info','ViewMesh'})
fe_cyclicb('Displaysel 2',cf,def,cf.Stack{'info','ViewMesh'})
```

3.2 Bearing and support representations

3.2.1 Linear bearing

One can simply represent a support or a bearing by a linear spring that is to say a `celas` element (for more details see `sdtweb('celas')`). A `celas` element is announced in the model element matrix by the header `[Inf abs('celas')]` and format is as follow: `[NodeId1 NodeId2 -DofID1 DofID2 ProID EltID Kv Mv Cv]` `NodeId1` and `NodeId2` define between which nodes `celas` is connected (if `NodeId2=0` celas is grounded), `DofID1` and `DofID2` which dof are connected, and `Kv Mv` and `Cv` give respectively stiffness mass and damping. A typical bearing for a rotor turning around x will then be defined by

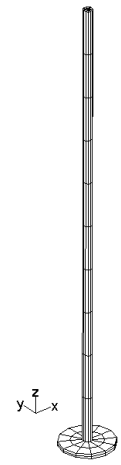

Figure 3.5: Simple disk on long rod model.

The following generates a spring bearing connected to node `n1`, stiffness is 1e9 and damping 1e2 in the xy plane. The use of a spring property is necessary for non linear time domain applications

```
model=d_rotor('TestDiskLongBeam');
n1=feutil('findnode z==.71 & x==0 & y==0',model);
model=feutil('AddElt',model,'celas',[n1 0   -12 0  100]);
model.il=p_spring(model.il,'dbval 100 1e10 0 1e2');
fe_simul('mode',model)

r1=struct('Origin',[20 10 0],'axis',[0 0 1],'radius',3, ...
          'planes',[1.5 1 111 1 3.1;
                    5.0 1 112 1 4;], ...
```

```
          'MasterCelas',[0 0 -123456 123456 10 0 1e14], ...
          'NewNode',0);
links={'connection','bearing',...
         struct('C1',1,'C2',r1,...
              'link',{{'celas',[0 0 23 0 103 0 1e5 0 0.1]}},...
              'NLdata',struct('type','RotCenter','lab','Bearing',...
                 'sel','ProId 1','JCoef',1,'drot',n1+.06))}
[model,RunOpt.idof]=nl_spring('ConnectionBuild',model,links)


[cf.mdl,RunOpt.idof]=nl_spring('ConnectionBuild',cf.mdl,links);
```

For 3d rotors, if there is no node on the axis of the shaft where one want to model the bearing, one can add a node whose displacement depends on displacements of nodes on a ring of the shaft using (see `sdtweb('fe_case#connectionscrew')`). Then one can connect a grounded celas to this node. xxx a full example is needed xxx

3.2.2 Non-linear bearings in the time domain

Documentation of `nl_spring` capabilities should go here.

3.3 Gyroscopic effects

SDT supports gyroscopic matrices in both rotating (body-fixed) and fixed frames (Eulerian representation of an axisymmetric structure). When performing assembly, the matrices in the local rotating frame are obtained with `MatType 7` gyroscopic and `MatType 8` centrifugal softening. `MatType 70` corresponds to the gyroscopic matrix in the global fixed frame. There is no centrifugal softening in this frame.

3.3.1 Fixed frame models

The `mass1` and `beam1` elements gyroscopic matrices are only available in the global fixed frame (`MatType 70`). The rotation axis is assumed to be the axis of the beam for `beam1` elements. Moments of inertia must be equal (axisymmetry). For

mass1 elements the rotation axis is assumed to be the one whose rotation inertia is different from the 2 others that must be equal.
mass1 .Elt format: [n mxx myy mzz ixx iyy izz EltId]
One can build simple models of 1d rotor using mass1 elements to represent rigid disk and beam1 to represent the shaft. One can find an example of such a rotor in d_rotor('TestShaftDiskMdl'). See section ?? for more details on how to mesh such a rotor.

For volume and shell elements, the formulation of gyroscopic matrices in global fixed frame is unclear and thus not currently implemented.

3.3.2 Rotating frame models

For all volume elements, one can compute gyroscopic (MatType 7) and centrifugal softening (MatType 8) matrices in the local rotating frame.

Elements under development are

- mass1 : Only point masses with same mass along the 3 translation DOF and no rotation inertia are considered. Other are ignored.
- beam1 : Not supported.
- shell : Not supported.

In that case rotation axis must be given as a vector in info,Omega stack entry in the model. For example model=stack_set(model,'info','Omega',[0 1 0]) will define a rotation axis along Y. Note that the norm of this vector is assumed to be the rotation speed in rad/s. The norm of the rotation vector should be 1 so that matrices are assembled for a rotation speed of 1 rad/s. Indeed it is assumed that gyroscopic and centrifugal softening matrices have been assembled for a rotation speed of 1 rad/s in many functions (such as fe_rotor etc. ...).

3.4 Frequency domain analysis, full model

Supported computations in fe_rotor, are

- Campbell diagram building

- Direct computation of critical speeds
- Frequency domain response to unbalance and harmonic loads on bearings
- Time domain simulation accounting for non-linear bearings is under development (these are performed with the shaft in a rotating frame and the stator fixed).

3.4.1 Campbell diagrams, full model

Campbell diagrams are implemented in the `fe_rotor Campbell` command.

```
model=d_rotor('TestShaftDiskMdl');
fe_rotor('Campbell -crit -cf100',model,linspace(0,8000,100));
figure(100);set(gca,'ylim',[0 200])
```

Command options are `-cfi` to define figure where to plot, `-crit` to compute critical speeds, `-stability` to display a stability diagram (damping-rotation speed).

The following is an example of a simple disk rotating at the end of a long rod. For other details see `rotor2d`.

```
% Model Initialization
mo1=rotor2d('test simpledisk -back'); % init model
cf=feplot;cf.model=rotor2d('buildFrom2D',mo1);
SE=cf.Stack{'disk1'}; % enforce boundary cond. on sector and assemble
SE=fe_case(SE,'FixDof','Base','z==1.01');
SE=fe_cyclic('assemble -se',SE);
cf.Stack{'disk1'}=SE;  fecom('view1');

% automated building of Campbell diagram : XXX NEED REVISION
RunOpt=struct('targ',1, ...
              'Range',linspace(0,1,30));
cf.def=rotor2d('teig',cf,RunOpt);
d1=fe_def('subdof',cf.def,feutil('findnode r==0',SE));
```

3.4.2 Blade with centrifugal stiffening

One considers stiffness matrices that are dependent on the rotation speed. Assuming that a second order polynomial representation is sufficient, one can build a

vector of weigthing coefficients

$$\begin{Bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{Bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ \Omega_1 & \Omega_2 & \Omega_3 \\ \Omega_1^2 & \Omega_2^2 & \Omega_3^2 \end{bmatrix}^{-1} \begin{Bmatrix} 1 \\ \alpha_2 \\ \alpha_3 \end{Bmatrix} \tag{3.1}$$

Such that the stiffness at rotation speed Ω is approximated by

$$[K(\Omega)] = \sum_{\alpha_i} [K\Omega_i] \tag{3.2}$$

The `zCoef` uses velocity `Omega` in rad/s.

This example now treats computation at variable rotation speeds

```
% Model Initialization
model=demo_cyclic('testblade');cf=feplot(model);
% Compute matrix coefficients for a multi-stage rotor
range=struct('data',[0 0 1;0 0 8e3;0 0 16e3],'unit','RPM');
% Assembling in the feplot figure, allows memory offload
fe_cyclicb('polyassemble -noT',cf,range);
X=struct('data',linspace(0,16e3,10),'unit','RPM');
fe_rotor('Campbell -cf1',cf.mdl,X);set(gca,'ylim',[0 3000])
```

Now a more standard non-linear static computation for a range or rotation speeds.

```
model=demo_cyclic('testblade');cf=feplot(model);
r2=linspace(1e3,15e3,20)';
range=struct('data',r2*[0 0 1],'unit','RPM');
model=stack_set(model,'info','Omega',range);
model=fe_cyclic('LoadCentrifugal',model)
opt=fe_simul('NLStaticOpt');
opt.MaxIter=100;opt.JacobianUpdate=1;
def=fe_time(opt,model);def.data(:,1)=range.data(:,3);
```

3.4.3 Complex modes

To compute complex modes at a given rotation speed, one can use `fe_rotor Campbell` command with option `-cmode`. Complexe modes are the computed using a

real mode basis projection. Mode options should be stored in the 'info' 'EigOpt' model stack entry. The complexe modes are return for the first rotation speed given as input argument.
Once modes are computed, you can display them in feplot. The fecom ShowTraj command displays trajectories. For example:

```
model=d_rotor('TestShaftDiskMdl')
model=fe_caseg('assemble -SE -secdof -matdes 2 1 70',model); % assemble model
def=fe_rotor('Campbell -cmode -cf100',model,100); % C modes at 100 RPM.
cf=feplot(model,def); fecom('view3');iimouse('resetvie');
fecom(cf,'ShowTraj')
```

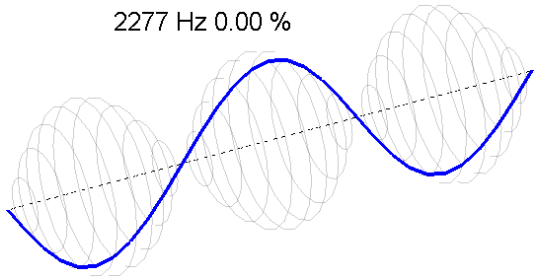

Figure 3.6: Complex mode and trajectory.

3.4.4 Forced frequency response to unbalanced load

One can compute the frequency response to an unbalanced load, or to an asynchronous load using fe_rotor RotatingLoad and ForcedResponse command (see sdtweb('fe_rotor') for more details). Definition of the load is different in local rotating frame or global fixed frame. One can see the validation part of this document for various examples. Following example computes the response to an unbalanced mass on the 1D rotor model:

```
mb=1e-4; db=0.15; % mass, distance to axis
om=linspace(0,6000,201); % RPM
model=fe_rotor(sprintf('RotatingLoad 2 %.15g -90 2',mb*db),model); % unbalance
r1=struct('Omega',om/60*2*pi,'w',om/60*2*pi); % Range
model=stack_set(model,'info','Range',r1);
R1=fe_rotor('ForcedResponse',model); % compute forced response
iiplot(R1) % plot response
```

3.5 Solvers for models with cyclic symmetry

The SDT/Rotor module contains

- classical cyclic symmetry solvers, where one assumes the solution to be associated with a specific number of diameters (spatial harmonic associated to the Fourier transform of a periodic geometry, see [5] for more details.)
- full rotor reduced models where cyclic symmetry solutions are used to build a reduced model for various stages. The associated solvers are discussed in section 3.6.

3.5.1 Static response

Resolution of static responses is performed using `fe_cyclicb ShaftSolve`. You should be aware, that non-linear static iterations may fail to converge if you have rigid body modes in your system. In the example below, this is the reason for fixing the axial motion of point 136 in disk1 and using the `-FixTan` which adds a tangential translation constraint on the first disk.

```
if 1==1 % One disk case
 cf=demo_cyclic('testrotor 7 -blade -cf 2');
 Sel={'disk1','groupall'};
else % Two disk case
 cf=demo_cyclic('testrotor 7 10 -blade -cf 2 reset');
 Sel={'','EltName SE';
      'disk1(1:2)','groupall';'disk2(1:3)','groupall'};
end
cf.Stack{'disk1'}=fe_case(cf.Stack{'disk1'},'FixDof','Axial',136.03);

% Linear static response
cf.Stack{'info','Omega'}=struct('data',250,'unit','RPM');
d0=fe_cyclicb('shaftsolvestatic 0 -FixTan',cf.mdl)
fe_cyclicb('displaySel',cf,d0,Sel)
fecom('ColorDataEvalRadZ')

% Non-linear static response
cf.Stack{'info','Omega'}=struct('data',250,'unit','RPM');
d0=fe_cyclicb('shaftsolvestatic 0 -FixTan -nlstep 1e-10',cf.mdl)
```

```
fe_cyclicb('displaySel',cf,d0,Sel)
fecom('ColorDataEvalRadZ')
```

XXX example with thermal loading xxx

Example with temperature dependent properties.

3.5.2 Single stage mode computations

As a first example one will consider computations of a single disk using mono harmonic solutions

Call `shaftTeig` in `fe_cyclicb`allows to compute the specified mono-harmonic normal modes.

Mono-harmonic normal modes are recovered to the rotor with the help from command `shaftdispay`. If command option `sel` is specified and a selection of elements is provided, the shapes are recovered to that selection only. Selections are cell arrays with the typical entry `Sel={'disk1',sel1}` where `sel1` is either a string (to select a subset of elements in the true mesh) or a list of elements (to build a reduced viewing mesh).

```
% Model Initialization
cf=demo_cyclic('testrotor 7 -blade -cf 2 reset');

% Mono-harmonic Solutions
model=stack_set(cf.mdl,'info','EigOpt',[5 5 -1e3 11 1e-8]);
def=fe_cyclicb('shaft Teig 0 5',model);

% partial display and computation of strain energy
Sel={'disk1','withnode {r<130}'};
fe_cyclicb('Displaysel',cf,def,Sel,'enerkdens');
if 1==2 % total display if needed
 cf.sel='groupall';cf.def=fe_cyclicb('Display',cf,def);
end

cf.Stack{'info','StressCritFcn'}='fe_cyclicb(''StressRR'');'
[dfull,STRESS]=fe_cyclicb('Displaysel',cf,def,Sel,'stress-gstate');
z=STRESS.GroupInfo{1,5};figure(1);plot(squeeze(z.Y(1,1,:,7)))
```

This will be extended to full disk computations in section **??**.

3.5.3 Multi-stage harmonic mode computations

Call shaftTeig in fe_cyclicb allow to compute the specified mono-harmonic solutions (multi-stage solutions for which disks share the same Fourier harmonic coefficient δ) in a single job.

Mono-harmonic eigensolutions are recovered using Dispay. If command option sel is specified and a selection of elements is provided, the shapes are recovered to that selection only. Selections are cell arrays with the typical entries:

- Sel={'disk1',sel1} where sel1 is either a string (to select a subset of elements in the true mesh) or a list of elements (to build a reduced viewing mesh),
- Sel={'',selg} where selg is a string that selects elements of the global mesh (selg is often 'eltname==SE' so that only disks are displayed).

In the same fashion, mono-harmonic static responses are returned by ShaftSolveStat This is of particular interest to compute the static deformation under the centrifugal loading (known to have a Fourier harmonic coefficient of $\delta = 0$) and built with command LoadCentrifugal within fe_cyclic.

```
% Model Initialization
cf=demo_cyclic('testrotor 7 10 -blade -cf 2 reset');

% Mono-harmonic Solutions
model=stack_set(cf.mdl,'info','EigOpt',[5 5 -1e3 11 1e-8]);
def=fe_cyclicb('shaft Teig 0 2',model);
demo_cyclic('RefcheckDisk710',def) % non regression check
cf.Stack{'disk1'}=fe_case(cf.Stack{'disk1'},'park','blade','innode{r>=20
Curve=fe_cyclicb('fourier 1:13 -mono -egyfrac',cf,def); % check energies
iiplot(Curve);colormap(flipud(hot));

cf.def=fe_cyclicb('Display',cf,def);

% static responses : sdtweb('freqcyclic#cyclic_static')
cf.Stack{'info','Omega'}=struct('data',1000,'unit','RPM');
cf.Stack{'curve','StaticState'}=fe_cyclicb('shaftsolvestatic 0',cf.mdl)

% Pre-stressed modes
dp=fe_cyclicb('shaft Teig 0 2',model);
```

```
[def.data dp.data]
```

After full rotor assembly restitution is performed using _SeRestit.

3.5.4 Campbell diagrams

First example of the beam with single disk

```
% Model Initialization
model2D=rotor2d('test simpledisk -back');
cf=feplot(rotor2d('buildFrom2D',model2D));
SE=cf.Stack{'disk1'}; % enforce boundary cond. on sector and assemble
SE=fe_case(SE,'FixDof','Base','z==1.01'); cf.Stack{'disk1'}=SE;

% Compute matrix coefficients for a multi-stage rotor
range=struct('data',[0 0 1; 0 0 1000/2/pi*60],'unit','RPM');
fe_cyclicb('polyassemble -noT',cf,range);

% Now run a mono-harmonic computation returning reduced model
cf.Stack{'info','Omega'}=struct('data',range.data(1,:),'unit','RPM');
cf.Stack{'info','EigOpt'}=[5 20 0]; % define eigenvalue options
MVR=fe_cyclicb('shafteig 1 -ReAssemble 2 -NoN -buildMVR',cf);
MVR.Stack{end}(end+[-1:0],4)={'1';'0'};  % skip problem with geometric softeni

rc=struct('data',linspace(0,8000*2*pi,50),'unit','RPM');
hist=fe_rotor('Campbell',MVR,rc);
fe_rotor('plotCampbell',hist,struct('fig',100,'axProp',{{'ylim',[0 250]}}))
```

Another example will be needed to treating the multi-stage case. This example needs further validation and rewrite.

```
% Model Initialization
model=demo_cyclic('testrotor 7 10 -blade');
cf=feplot(model);

% Compute matrix coefficients for a multi-stage rotor
range=struct('data',[0 0 1;0 0 800;0 0 1600],'unit','RPM');
% Assembling in the feplot figure, allows memory offload
model=fe_cyclicb('shaftRimAsSe',model); % Needed for PolyAssemble
```

```
fe_cyclicb('polyassemble -noT',cf,range);

% Now run a mono-harmonic multi-speed computation
cf.Stack{'info','Omega'}=struct('data',range.data(1,:),'unit','RPM');
cf.Stack{'info','EigOpt'}=[5 20 -1e3]; % define eigenvalue options
MVR=fe_cyclicb('shafteig 1 -ReAssemble 2 -NoN -buildMVR',cf);

%MVR.Stack{end}(end+[-2:0],4)={'1';'0';'0'};  % skip problem with geomet

rc=struct('data',linspace(1,8000,50),'unit','RPM');
hist=fe_rotor('Campbell',MVR,rc);
fe_rotor('plotCampbell',hist,struct('fig',100,'axProp',{{'ylim',[0 3e3]}

%Sel={'disk1','groupall';'disk2','groupall'};
%fe_cyclicb('Displaysel',cf,def,Sel)
```

3.5.5 Complex modes

Need documentation here.

3.5.6 Forced frequency response to unbalanced load

Need documentation here.

3.6 Full rotor model from cyclic computation

3.6.1 Single stage full rotor example

Starting from the mono-harmonic computation in section 3.5.2. One builds a full shaft model that will allow prediction of all the modes.

```
%H5.close;fclose all; % may be needed for overwrite
cf=demo_cyclic('testrotor 7 -blade -cf 2');
cf.Stack{'info','Omega'}=struct('data',250,'unit','RPM');
```

```
d0=fe_cyclicb('shaftsolvestatic 0',cf.mdl) % auto-display with no arg
cf.Stack{'curve','StaticState'}=d0;
RunOpt.Root=fullfile(sdtdef('tempdir'),'Disk_7');
RunOpt.FileSe=fullfile(sdtdef('tempdir'),'Disk_7_SE');

cf.mdl=fe_cyclicb('shaftSeAssemble -reset',cf.mdl,RunOpt.FileSe);
RunOpt.FileTeig=fullfile(sdtdef('tempdir'),'Disk_7_TEIG.mat');

% See sdtweb('fe_cyclicb#ShaftEig') for -batch option
def=fe_cyclicb('shaft Teig 0 1 5',cf.mdl);

% Now build a multi-harmonic model
fe_cyclicb('ShaftPrep -handle',cf,def);
fesuper('fassemble',cf);

% Mode Computations
defr=fe_eig(cf.Stack{'MVR'},[5 50 1e3 11 1e-8]);
cf.sel='reset';cf.def=fesuper('sedef',cf,defr);

def_ext=fe_cyclicb('DefList',RunOpt.FileTeig(1:end-4))

% the same results should be achived by assembling the prestress inside
% shaftTeig xxx update matrices only once, not for each diameter
cf=demo_cyclic('testrotor 7 -blade -cf 2');
cf.Stack{'info','Omega'}=struct('data',250,'unit','RPM');
cf.mdl.il=[1001 fe_mat('p_super','SI',1) 1 0 0 1];

fe_cyclicb('shaft Teig 0 1 5 -batch -reassemble',cf.mdl,RunOpt.FileTeig);
def_int=fe_cyclicb('DefList',RunOpt.FileTeig(1:end-4))

%if norm(def_int.data(def_int.data(:,1)>1,1)-def_ext.data(def_ext.data(:,1)>1,
%error('ShaftTEig internal and external prestress produce different results');
```

Note that to build a disk model from a sector, you should use `model=fe_cyclicb('DiskFromS 1',model,sector)`. xxx Arnaud : need to check adaptation for multi-disk models.

3.7 Time domain analysis

Once rotor model is created, one can perform time computation. nl_spring

function is very useful in that case. Gyroscopic effects can be treated as non linear load depending on instant value of a model DOF (observation of the rotation speed for example). Links between fixed and rotating parts can also be modelized as non linear loads.

3.7.1 Simple example

Following example simply computes the response to an impact on the disk, assuming that rotation speed is 1000 RPM:

```
model=d_rotor('TestShaftDiskMdl'); % build simple 1D rotor

% Define Time option and other parameters:
opt=fe_time('TimeOpt Newmark .25 .5 0 1e-4 1e4'); % TimeOpt
Range=struct('Omega',1000/60*2*pi);
model=stack_set(model,'info','Range',Range);
opt.matdes=[2 1 70];
model=stack_set(model,'info','TimeOpt',opt);
model=fe_rotor('TimeOptAssemble',model);

% initial impact:
model=fe_case(model,'DofLoad','In',...
    struct('def',1,'DOF',2.02,...
           'curve',sprintf('TestStep%.15g',opt.Opt(4)*5)));

% Compute response:
def=fe_time(model); iiplot(def);
```

3.7.2 Gyroscopic effects

Gyroscopic coupling is represented by a load $-\Omega(t).D(\Omega = 1) * V$ where Ω is the rotation speed and D the gyroscopic coupling matrix. That can be applied at each step of time using the `DofKuva` `nl_spring` non linearity (see `sdtweb('nl_spring')`).

An unbalanced mass is a load proportional to rotation speed. In the local rotating frame it can be described using `nl_spring` `DofV` non linearity.

Following example deals with the simple 1D rotor and performs a time integration in fixed global frame taking in account the gyroscopic effect, for an initial impact on the disk. Note that the non linear holding of the gyroscopic effect is not necessary here since the global frame is considered, and rotation speed is assumed

to be constant (1000 RPM). The gyroscopic effects (in green in figure below) are coupling the y and z motion.

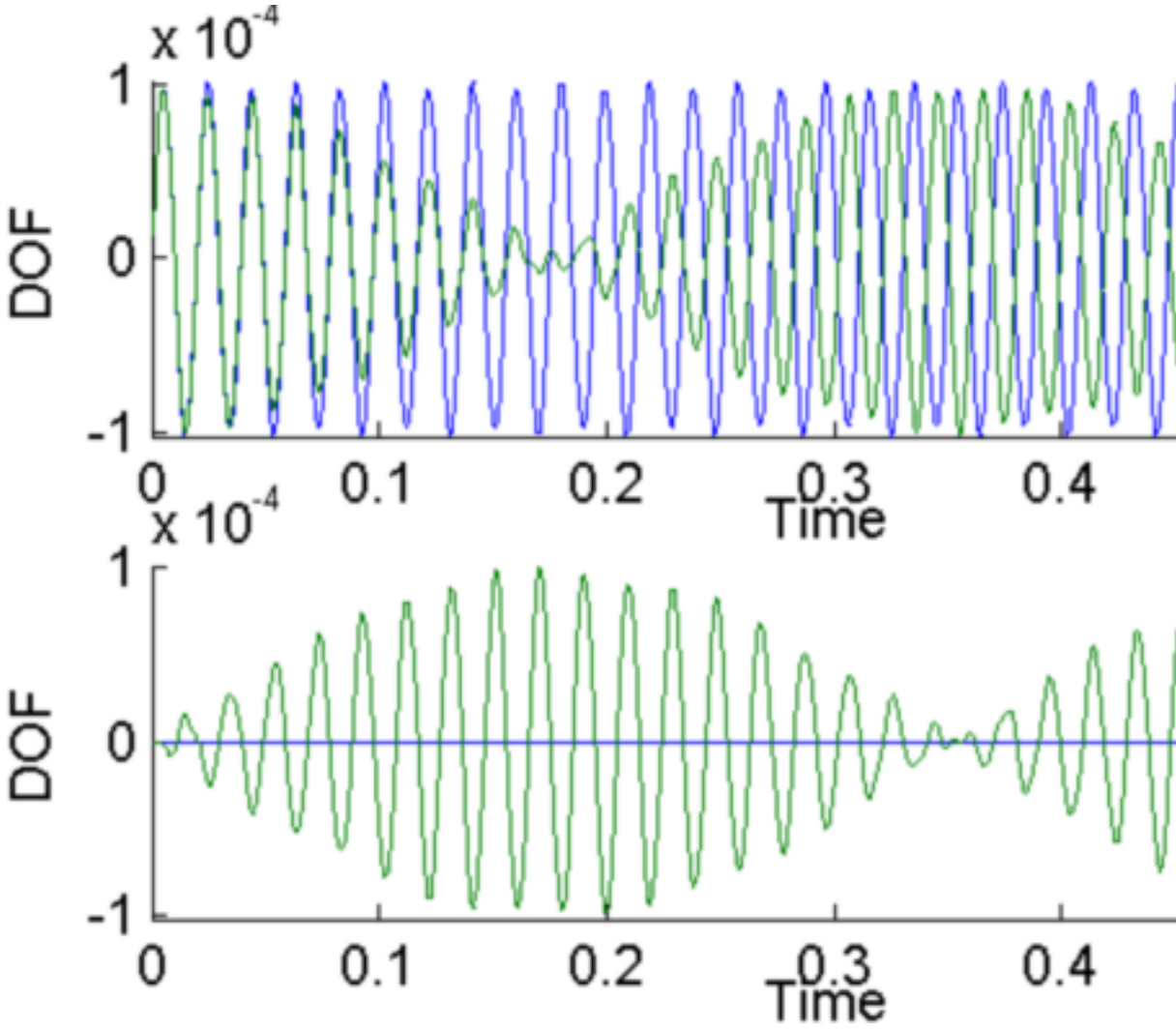

Figure 3.7: y (top) and z (bottom) deflection, with (green) and without (blue) gyroscopic effect.

```
model=d_rotor('TestShaftDiskMdl');
model=fe_case(model,'FixDof','Ends',...
```

```
          [1.01 1.02 1.03 4.01 4.02 4.03 0.01]);
% TimeOpt:
opt=nl_spring('TimeOpt -gamma .501');
opt.NeedUVA=[1 1 1];
opt.IterEnd='if ite>90;evalin(''caller'',''assignin(''''base'''',''''def
opt.Follow=1; opt.RelTol=-1e-5;
opt.Opt(7)=-1; % factor type sparse
opt.Opt(4)=1e-4; opt.Opt(5)=0.6e4; % NSteps
% Initial impact:
model=fe_case(model,'DofLoad','In',...
    struct('def',1e3,'DOF',2.02,...
           'curve',sprintf('TestStep%.15g',opt.Opt(4)*5)));
% Without gyroscopic effect:
def0=fe_time(opt,model) % compute

% With NL gyroscopic effects:
model=stack_set(model,'pro','DofKuva1005', ... % gyroscopic effects
    struct('il',[1005 fe_mat('p_spring','SI',1) 0 0 0 0 0],...
    'type','p_spring','NLdata',struct(...
        'type','DofKuva','lab','gyroscopic effect', ...
        'Dof',[],'Dofuva',[0 1 0],'MatTyp',70,...
        'factor',-1*1000/60*2*pi,'exponent',1,'uva',[0 1 0])));

def=fe_time(opt,model) % compute

% Display results as curves
ci=iiplot;
iicom('sub 2 1');
iicom(ci,'IIxOnly',{'disp(1)';'disp(2)'});
i1=fe_c(ci.Stack{'disp(1)'}.DOF,2+[.02;.03],'ind');
iicom(ci,sprintf(';cax1;chc%i;cax2;chc%i',i1(1),i1(2)));
% comgui('ImWrite',comgui('cd','o:\sdt.cur\rotor\plots\shaftdisk_withand
% display deformation
cf=feplot(model); % display model
cf.def=def0; % without gyrocopic effects
cf.def=def;  % with gyroscopic effects
```

3.7.3 Other representations of bearings

In the fixed basis the simplest bearing model is a linear spring (celas element. In rotating basis, using a spring to model beaing is not possible using a celas element. The nl_spring RotCenter non linearty is usefull to modelize link between rotating and non rotating parts.

In the fixed basis one can use a non linear spring to model the bearing. The nl_maxwell non linearity describes a set of generalized maxwell rheological models that can be used as non linear bearing in the global frame. As an illustration, following example replaces the linear bearing of the 1D rotor by a zener model of the link.

```
model=d_rotor('TestShaftDiskMdl'); % Build model
model=fe_case(model,'FixDof','Ends',...
                    [1.01 1.02 1.03 4.01 4.02 4.03 0.01]); % Clamp ends
% TimeOpt:
opt=nl_spring('TimeOpt');
opt.Follow=1; opt.RelTol=-1e-5;
opt.Opt(7)=-1; % factor type sparse
opt.Opt(4)=1e-4; opt.Opt(5)=0.6e4; % NSteps
opt.IterInit='model.FNL0=model.FNL+0;';
% Initial impact:
model=fe_case(model,'DofLoad','In',...
    struct('def',1e3,'DOF',2.02,...
           'curve',sprintf('TestStep%.15g',opt.Opt(4)*5)));
% NL bearings
% define nl_maxwell data
model.il(ismember(model.il(:,1),[100;101]),:)=[]; % remove properties
k0=500000; k1=k0/2; c1=600; % 20.41% for f=54.15 Hz
data=nl_maxwell(sprintf('db zener k0 %.15g  k1 %.15g c1 %.15g',k0,k1,c1));
data.Sens{2}=3.02; % translation sensor defining nl_maxwell inputs
r1=p_spring('default'); r1=feutil('rmfield',r1,'name');
r1.NLdata=data; r1.il(3)=k0; % translation sensor defining nl_maxwell inputs
r1.il(1)=100; model=stack_set(model,'pro','bearingy',r1);
r1.NLdata.Sens={'trans',3.03}; % translation sensor defining nl_maxwell inputs
r1.il(1)=101; model=stack_set(model,'pro','bearingz',r1);

ci=iiplot(3); % results will be store there
def0=fe_time(opt,model); % compute
```

4

Validation

Contents

4.1 Rigid disk example 44
4.1.1 Matrices in rotating frame 44
4.1.2 Matrices in global fixed frame 45
4.1.3 Validation with 3D model disk 47
4.2 Simple 2DOF model of shaft with disk 50
4.3 1D models 54
4.3.1 1D example in a fixed frame 54
4.3.2 1D models in a rotating (body-fixed) frame 57
4.4 3D rotor 57
4.5 Data structure reference 59

4.1 Rigid disk example

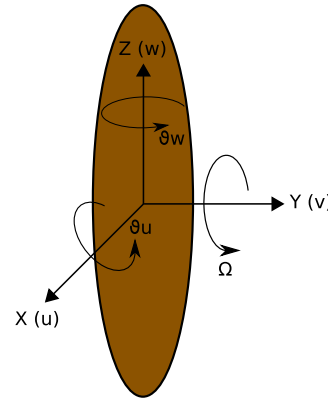

Figure 4.1: Simple rigid disk.

We consider a simple rigid disk of axis Y, thickness h, radius R, mass m. This simple example is very useful because we can easily compute matrices in both of global and local frame for a simple description of motion with 4 DOF. Besides we can compute in SDT equivalent matrices for a `mass1` rigid disk, for a disk described by a `beam1` element and for a volume disk in `hexa8` elements. Then we can compare gyroscopic matrices and make sure that their implementation for each element is correct.

4.1.1 Matrices in rotating frame

The motion of the disk is described by 2 translations (u_c and w_c) and 2 rotations DOF (θ_u and θ_w). Disk is assumed to be rigid so displacements at each point of the disk are given by the shape functions

$$\left\{ \begin{array}{c} u \\ v \\ w \end{array} \right\} = \left[\begin{array}{cccc} 1 & 0 & 0 & -v \\ 0 & 0 & -w & u \\ 0 & 1 & v & 0 \end{array} \right] \left\{ \begin{array}{c} u_c \\ w_c \\ \theta_u \\ \theta_w \end{array} \right\} = [N_{xyz}] \{q\} \tag{4.1}$$

Rotation matrix (2.3) along Y axis is given by

$$[\Omega] = \Omega \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{4.2}$$

Using matrix expression given in section 2.1.1 in body fixed local rotating frame, and projecting the integrals by assuming rigid disk motion

$$[M] = \begin{bmatrix} m & 0 & 0 & 0 \\ 0 & m & 0 & 0 \\ 0 & 0 & I_u & 0 \\ 0 & 0 & 0 & I_w \end{bmatrix} \tag{4.3}$$

with $I_u = I_w = \frac{1}{4}mR^2 + m\frac{h^2}{12}$

$$[D] = 2m\Omega \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{h^2}{12} \\ 0 & 0 & -\frac{h^2}{12} & 0 \end{bmatrix} \tag{4.4}$$

$$[Kc] = -m\Omega^2 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{h^2}{12} & 0 \\ 0 & 0 & 0 & \frac{h^2}{12} \end{bmatrix} \tag{4.5}$$

$$[M]\{\ddot{q}_L\} + D\{\dot{q}_L\} + Kc\{q_L\} = F_L \tag{4.6}$$

An unbalanced mass is represented by a static load in the local rotating frame

$$\{F_L\} = \begin{Bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \tag{4.7}$$

4.1.2 Matrices in global fixed frame

Local frame (indexed with $_L$) is rotating in global fixed frame (indexed with $_G$) according to rotation matrix

$$[R_t] = \begin{bmatrix} cos(\Omega t) & -sin(\Omega t) \\ sin(\Omega t) & cos(\Omega t) \end{bmatrix} \tag{4.8}$$

so that we have
$\{q_L\} = [R_t]\{q_G\}$
$\{\dot{q}_L\} = [R_t]\{\dot{q}_G\} + \left[\dot{R}_t\right]\{q_G\}$
$\{\ddot{q}_L\} = [R_t]\{\ddot{q}_G\} + 2\left[\dot{R}_t\right]\{\dot{q}_G\} + \left[\ddot{R}_t\right]\{q_G\}$

and equation (4.6) can thus be rewritten as

$$\begin{gathered} \left[R^T M R\right]\{\ddot{q}_G\} + \left[2R^T M\dot{R} + R^T D R\right]\{\dot{q}_G\} + \left[R^T M\ddot{R} + R^T D\dot{R} + R^T K_c R\right]\{q_G\} = F_G \\ [M_G]\{\ddot{q}_G\} + [D_G]\{\dot{q}_G\} + [Kc_G]\{q_G\} = F_G \end{gathered} \tag{4.9}$$

One has $R^T\dot{R} = \Omega\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ and $R^T\ddot{R} = \Omega^2\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ thus in the gyroscopic effect the translation terms disappear and one has

$$[D_G] = 2\Omega\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -I_v \\ 0 & 0 & I_v & 0 \end{bmatrix}$$

with $I_v = \frac{1}{4}mR^2 = I_u - 2mh^2/12$ and the xxx UNEXPECTED xxx centrifugal softening in the global frame

$$[Kc_G] = -\Omega^2\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2}I_v & 0 \\ 0 & 0 & 0 & \frac{1}{2}I_v \end{bmatrix}$$

An unbalanced mass is represented by a rotating load in the global fixed frame

$$\{F_G\} = \begin{Bmatrix} sin(\Omega t) \\ cos(\Omega t) \\ 0 \\ 0 \end{Bmatrix} \tag{4.10}$$

4.1.3 Validation with 3D model disk

This validation example considers a disk meshed with hexa8 volume elements. Local frame matrices are computed and projected on the geometrical rigid body modes (x and z translations, and corresponding rotation). One checks matching of

- matching of theoretical and numerical matrices
- response amplitudes of the disk with unbalanced mass in global and local frame
- mass1 and beam1 elements gives the same gyroscopic matrices in the global frame (mattype 70)

```
% model of disk:
R1=.01;R2=.15;h=0.03;rho=7800;
md=d_rotor(sprintf('testvoldisk -dim %.15g %.15g %.15g 5 24',R1,R2,h))
md.DOF=[];md=fe_caseg('assemble -se -matdes 2 1 7 8 not',md);
% disk assumed to be rigid:
rb=feutil('geomrb',md); cf=feplot(md); cf.def=rb;
md.K=feutil('tkt',rb.def(:,[1 3 4 6]),md.K);
md.DOF=[1.01 1.03 1.04 1.06]';
md.K{2}=0*md.K{2}; % rigid disk
md.K{2}(1,1)=5e5;md.K{2}(2,2)=5e5; % bearing
md.K{2}(3,3)=1e5;md.K{2}(4,4)=1e5; % bearing rot /XXX

% Campbell in local rotating frame:
fe_rotor('campbell-critical',md,linspace(0,9000,31));
% Unbalanced mass:
w=1; t=12; wrange=logspace(0,3,100); Q=[];
for j1=1:length(wrange);w=wrange(j1);
 % rotation matrix
 R=[cos(w*t) -sin(w*t); sin(w*t) cos(w*t)]; R=[R zeros(2);zeros(2) R];
 Rp=w*[-sin(w*t) -cos(w*t); cos(w*t) -sin(w*t)];Rp=[Rp zeros(2);zeros(2)
 Rpp=-w^2*R;
 % Global matrices
 MG=R'*md.K{1}*R;
 DG=2*R'*md.K{1}*Rp + R'*w*md.K{3}*R;
 KG=R'*md.K{1}*Rpp + R'*w*md.K{3}*Rp + R'*w^2*md.K{4}*R   + R'*md.K{2}*R;
 % compare to theoretical values
 m=pi*(R2^2-R1^2)*3e-2*7800;
```

```
 Iv=0.5*m*(R2^2+R1^2); Iu=0.5*Iv+m*h^2/12;
 DGth=zeros(4); DGth(4,3)=Iv; DGth(3,4)=-Iv;  DGth=w*DGth;
 KGth=diag([0 0 1 1]); KGth=-w^2*Iv/2*KGth;
 if norm(KG-R'*md.K{2}*R-KGth)/norm(KGth)>5e-2;error('Cent. soft mismatch');end
 if norm(DG-DGth)/norm(DGth)>5e-2; error('gyro matrix mismatch'); end
 % unbalanced mass:
 X0L=inv(md.K{2}+w^2*md.K{4})*w^2*[1 0 0 0]';  % local
 X0L=abs(X0L(1));
 X0G=inv(-w^2*MG+1i*w*DG+KG) *w^2*[-1i 1 0 0]'; % global
 X0G=abs(X0G(1));
 Q(1,j1)=X0L;  Q(2,j1)=X0G;
end
figure(4);semilogy(wrange*60/2/pi,Q(1,:),wrange*60/2/pi,Q(2,:));
legend('local','global'); title('Unbalanced mass');setlines
xlabel('Rotation velocity [RPM]'); ylabel('Amplitude [m]')
if norm(Q(1,:)-Q(2,:))/norm(Q(1,:))>1e-8; error('Global/Local error'); end

% check mass1 gyroscopic matrix 70:
mdmass=struct('Node',[1 0 0 0 0 0 0], ...
              'Elt',[Inf abs('mass1') 0; 1 m m m Iu Iv Iu]);
mdmass=fe_case(mdmass,'fixdof','Base',[1.02 ;1.05])
mdmass=fe_caseg('assemble -se -matdes 2 1 70',mdmass)
if norm(w*mdmass.K{3}-DGth)/norm(DGth)>1e-8; sdtw('_err','mass1 gyro 70 unmatch
% check beam1 gyroscopic matrix 70:
Omega=[0 1 0]; % axis of the disk
mdbeam1=struct('Node',[1 0 0 0 -h/2*Omega; 2 0 0 0  h/2*Omega],...
               'Elt',[Inf abs('beam1') 0; 1 2 1 1 0 0 0]);
mdbeam1.il=p_beam('convertto1',p_beam(sprintf('dbval 1 TUBE %.15g %.15g',R2,R1)
mdbeam1.pl=m_elastic('dbval 1 steel');
mdbeam1=fe_case(mdbeam1,'fixdof','fix1',...
 [1+(find(Omega)+[0;3])/100;2+(find(Omega)+[0;3])/100]);
rb=feutil('geomrb',mdbeam1); mdbeam1=fe_case(mdbeam1,'DofSet','fix2',rb);
rb=fe_simul('static',mdbeam1); mdbeam1=fe_case(mdbeam1,'remove','fix2');
mdbeam1.DOF=[]; mdbeam1=fe_caseg('assemble -se -matdes 2 1 70',mdbeam1);
mdbeam1.DOF=fe_case('gettdof',mdbeam1);
rb=feutil('placeindof',mdbeam1.DOF,rb)
mdbeam1.K=feutil('TKT',rb.def(:,setdiff(1:6,find(Omega)+[0 3])),mdbeam1.K)
if norm(w*mdbeam1.K{3}-DGth)/norm(DGth)>1e-8;
 sdtw('_err','beam1 gyro 70 unmatch');
```

```
end
```

The response to the unbalanced mass is the same in both of the 2 frames. The maximum of response matches rotation speed found as critical speed for forward whirl in local and global Campbell diagram.
Note that we can compute Campbell in local frame from Campbell in global frame with:
$f_L^{(F)} = \|f_G^{(F)} - \Omega\|$ for forward modes and
$f_L^{(B)} = \|f_G^{(B)} + \Omega\|$ for backward modes.
The 2 Campbell diagrams have been computed using matrices in corresponding frame and we check that we can pass from one to other using these formulas.

XXX In this example:
- Centrifugal softening in global fixed frame? Not 0
- In GLOBAL frame gyroscopic matrix DG do not modify unbalanced response because only translation dof are affected... - In LOCAL frame gyroscopic D do not modify unbalanced response: `X0L=inv(md.K2+`w^2`*md.K4)*`w^2`*[1 0 0 0]'; % local`

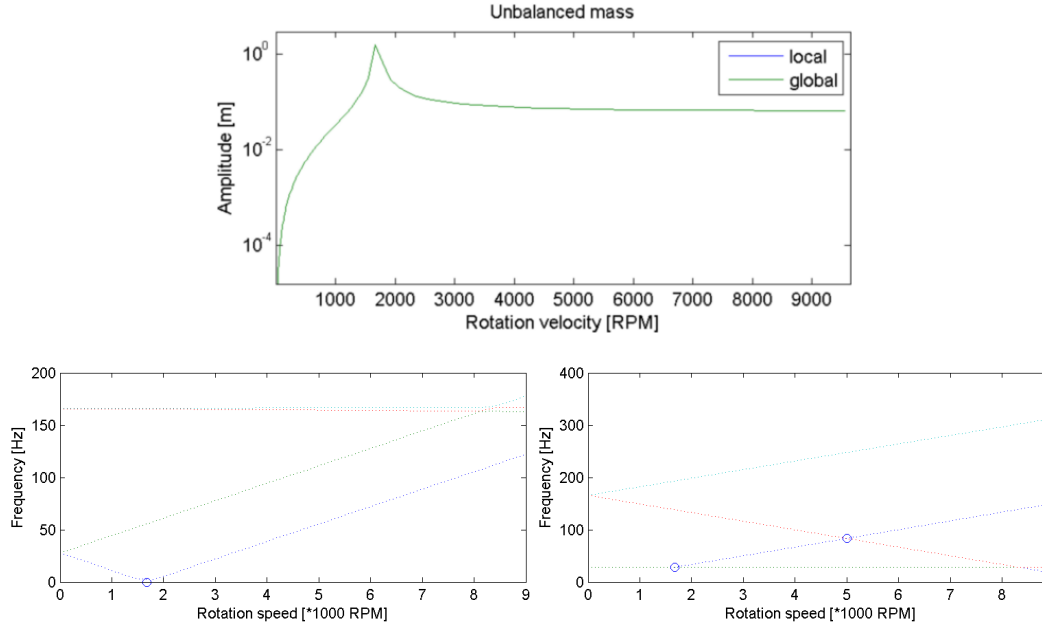

Figure 4.2: Top: Unbalanced mass response amplitude computed in local and global frame. Bottom left: Campbell diagram in local rotating frame, right: in global frame.

4.2 Simple 2DOF model of shaft with disk

This section is about the simple 2 DOF rotor model described in chapter 2 of Lalanne and Ferraris [7].

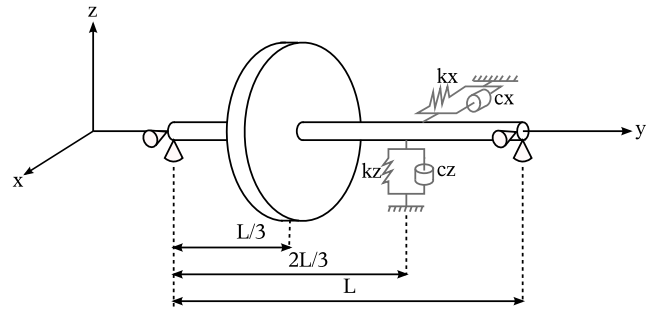

Figure 4.3: Simple model of rotor.

The disk is rigid, the shaft is described by 2 sin shape functions $f(y) = sin(\frac{\pi y}{L})$ where L is the length of the shaft.
Bearing at $y = \frac{L}{3}$ is represented by 2 additional stiffness and damping. $K_{bearing} = sin(\frac{2\pi}{3}) \begin{bmatrix} kxx & kxz \\ kzx & kzz \end{bmatrix}$ $C_{bearing} = sin(\frac{2\pi}{3}) \begin{bmatrix} cxx & cxz \\ czx & czz \end{bmatrix}$

The 2 DOF are considered in the global fixed frame.

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \{\ddot{q}\} + \Omega \begin{bmatrix} 0 & -a \\ a & 0 \end{bmatrix} \{\dot{q}\} + \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix} \{q\} = F \tag{4.11}$$

For an unbalanced mass mb at a position of L/3 on the shaft and at a distance of db of the rotation axis $F = m^* db \Omega^2 \begin{Bmatrix} sin(\Omega t) \\ cos(\Omega t) \end{Bmatrix}$ where $m^* = sin(\frac{\pi}{3}) mb$.

For a simple asynchronous load of amplitude f0 rotating at $s\Omega$ $F = f0 \begin{Bmatrix} sin(s\Omega t) \\ cos(s\Omega t) \end{Bmatrix}$.

Frequency analysis is performed.
Harmonic solutions are computed under the form $\Re(Q_0 exp(\imath s\Omega t))$ ($s = 1$ for unbalanced mass and $s <> 1$ for asynchronous load).

Implementation of this case can be found in `d_rotor Lalanne2DOF`. One can

define values of parameters (kxx, czx, m, a, mb, f0, s etc. ...) as fields of a data structure `r1` given as argument : `d_rotor('Lalanne2DOF',r1)`. A campbell and frequency forced response are then computed at the rotation speeds defined (in RPM) in the field `.om` of the parameter data structure. One can ask only for the model using `model=d_rotor('Lalanne2DOFMdl',r1);`.

If this first case, no bearing is considered ($K_{bearing} = 0$ and $C_{bearing} = 0$).

```
beta=0;
r1=struct('om',linspace(0,9e3,101),'s',1,...
      'kxx',0,'kzz',0,'kxz',0,'kzx',0,...
      'cxx',1e2*beta,'czz',5e2*beta,'cxz',0,'czx',0);
R1=d_rotor('TestLalanne2DOF',r1)
```

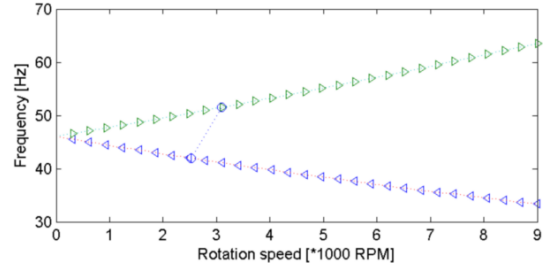

Figure 4.4: Campbell diagram. Symmetric rotor without bearing.

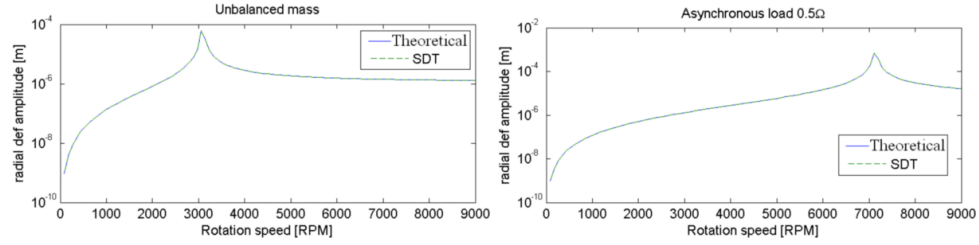

Figure 4.5: Response to an unbalanced load (left) and asynchronous load (right). Symmetric rotor without bearing.

Maximum response for unbalanced mass is obtained for the exact critical rotation speed of forward mode computed in the Campbell (3089 RPM). This speed is

the same as in Lalanne theoretical expression.

Bearing stiffness along z is now taken in account so that rotor is not symmetric. $kzz = 5e5$ and $kxx = kxz = kzx = 0$. There is no damping in the bearing: $C_{bearing} = 0$).

```
beta=0;
r1=struct('om',linspace(0,9e3,101),'s',1,...
     'kxx',0,'kzz',5e5,'kxz',0,'kzx',0,...
     'cxx',1e2*beta,'czz',5e2*beta,'cxz',0,'czx',0);
R1=d_rotor('TestLalanne2DOF',r1)
```

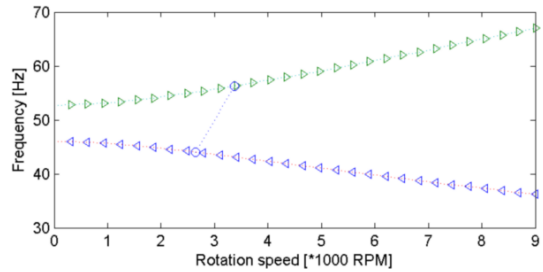

Figure 4.6: Campbell diagram. Asymmetric rotor with z stiffness bearing.

Now the 2 modes are not at the same frequency for $\Omega = 0$.

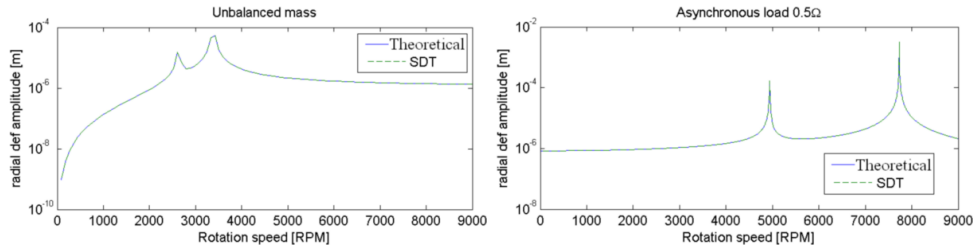

Figure 4.7: Response to an unbalanced load (left) and asynchronous load (right). Asymmetric rotor with z bearing stiffness.

Now there are 2 maximums of response that match critical rotation speeds for

both forward and backward modes. This is the same for the asynchronous load.

Bearing stiffness along z is taken in account so that rotor is not symmetric. $kzz = 5e5$, $kxx = 1e5$ and $kxz = kzx = 0$. Damping in the bearing is also considered with $cxx = 100\beta$, $czz = 500\beta$ and $cxz = czx = 0$.

```
beta=15;
r1=struct('om',linspace(0,20e3,201),'s',1,...
      'kxx',1e5,'kzz',5e5,'kxz',0,'kzx',0,...
      'cxx',1e2*beta,'czz',5e2*beta,'cxz',0,'czx',0);
d_rotor('TestLalanne2DOF',r1)
```

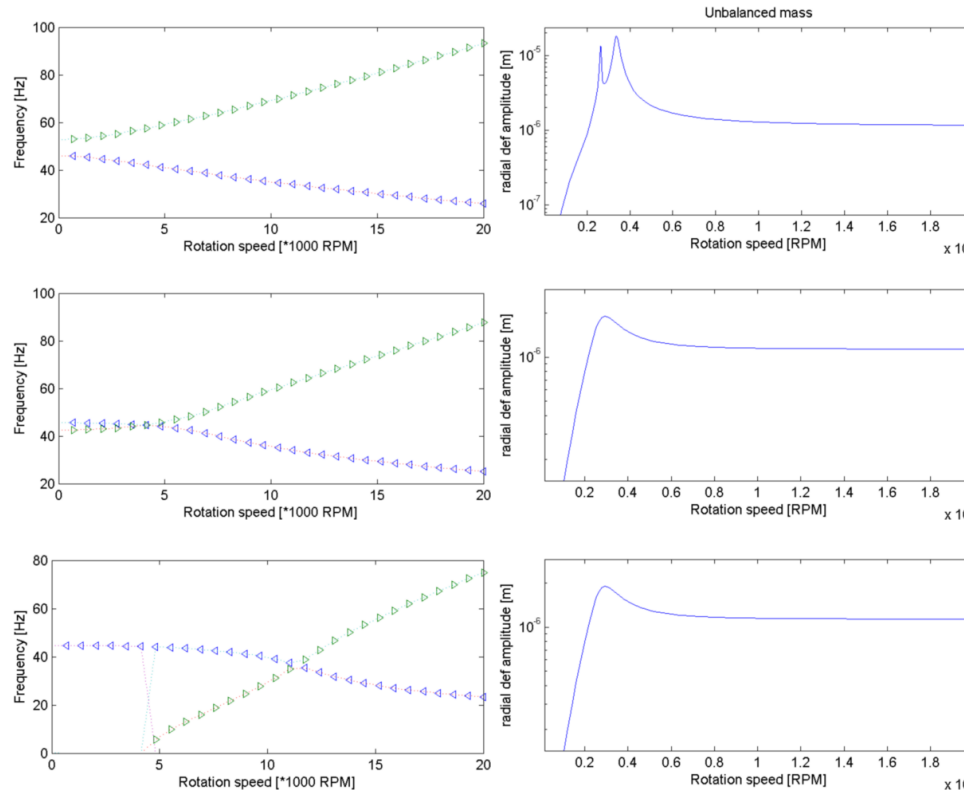

Figure 4.8: Campbell (left) and Responses to unbalanced load (right) for different damping (top:$\beta = 1$, middle:$\beta = 15$ and bottom:$\beta = 26$). Asymmetric rotor with z bearing stiffness and damping.

Backward and forward mode can cross each other in the Campbell diagram. Asymmetry leads to the excitation of the backward mode. Damping leads to a more spread resonance response.

4.3 1D models

4.3.1 1D example in a fixed frame

This first example treats the simple case, taken from [6], of a shaft with a rotating disk at one third the length.

In order to compare this model to the simple 2 DOF model of Lalanne (see section 4.2) we project the matrices on the 2 sin shape function of the shaft. Relative error for mass matrix is 0.01%, 4.59% for stiffness and 1.64% for gyroscopic matrix. Campbell for the projected model and the Lalanne 2 DOF model are almost the same. For the full 1d model (not projected) the increasing of the frequency of 1rst forward whirl mode tends to an asymptote.

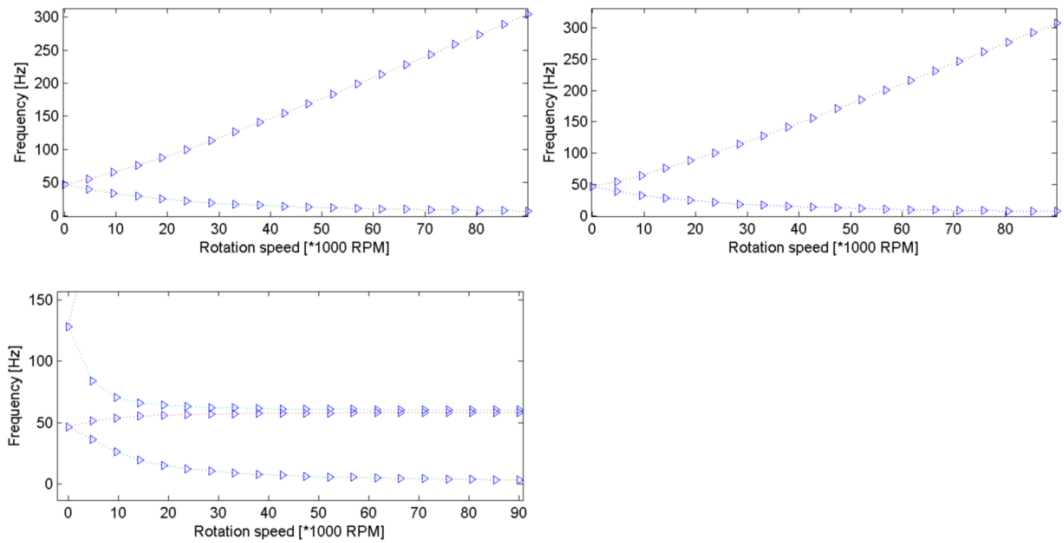

Figure 4.9: Campbell diagrams. Left: 1D rotor (up: projected on 2 DOF, down: full model), right: Lalanne 2 DOF rotor.

Following example performs preceding comparison :

```
% - LALANNE 2DOF
model2DOF=[]; m=14.29; k=1.195e6; a=2.871;
kxx=0; kzz=0; kxz=0; kzx=0;% bearing
% define model matrices :
model2DOF.K={m*eye(2), k*eye(2)+sin(pi*2/3)^2*[kxx kxz;kzx kzz],a*[0 -1;1
model2DOF.DOF=1+[.01;.03];
model2DOF.Klab={'m','k','Dg'};model2DOF.Opt=[1 0 0;2 1 70];
% rotate model so that x=rotating axis:
model2DOF.K=feutil('tkt',[1 0 0; 0 0 1],model2DOF.K);
model2DOF.K=feutil('tkt',[0 -1 0;1 0 0; 0 0 1],model2DOF.K);
for j1=1:3; model2DOF.K{j1}=model2DOF.K{j1}([2 3],[2 3]); end
model2DOF.DOF=[1.02;1.03];
model2DOF=stack_set(model2DOF,'info','Omega',[1 0 0]);

% - SHAFTDISK beam1+mass1  gyro 70 - - - - - - - -
 [model,TR]=d_rotor('TestshaftdiskMdl -TRLalanne')
 model.Elt=feutil('removeelt eltname celas',model);
 % project matrices on 2 sin shape functions equivalent to lalanne 2DOF m
 model=fe_caseg('assemble -secdof -matdes 2 1 70 3',model);
 TR=feutilb('placeindof',model.DOF,TR);
 model.K=feutil('TKT',TR.def,model.K); model.DOF=[2.02;2.03];
 % - compare with reference matrices from Lalanne
 for j1=1:3 % Compare matrices [model.K{j1} model2DOF.K{j1}]
  if normest(model.K{j1}-model2DOF.K{j1})/normest(model.K{j1})>0.05;
  sdtw('_err','2DOF and shaftdisk unmatch for mat %s', model.Klab{j1});
  end
 end
% Campbell with SDT matrices reduced on Lalanne shapes
r1=struct('data',linspace(0,9e3),'unit','RPM');
fe_rotor('campbell -nodir -crit -cf1',model,r1);

% - Now compare full beam model and 2D shapes
model.DOF=[]; model.K={};if ishandle(1);close(1);end
fe_rotor('campbell -nodir -crit -cf1',model2DOF,r1);
set(findall(1,'type','line'),'color','k', ...
 'linewidth',2,'linestyle','--');
hold on;
fe_rotor('campbell -nodir -crit -cf1',model,r1);
```

```
hold off;set(gca,'ylim',[0 160]);
h=findall(1,'type','line');legend(h(end+[0 -3]),'2 DOF','beam')
```

Following example computes frequency response to unbalanced or asynchronous load:

```
model=d_rotor('TestShaftDiskMdl'); % Model Initialization

% Assemble nominal matrices:
model=fe_caseg('assemble -reset -secdof -matdes 2 1 70',model);

% Campbell diagram and critical speeds:
fe_rotor('campbell -full -critical',model,linspace(0,20000,30));
set(gca,'ylim',[0 350]);

% Unbalanced mass or asynchronous load :
mb=1e-4; db=0.15; % mass, distance to axis
s=1; f0=1; % s=1, unbalanced load. s<>1, asynchronous load
om=sort([3296 3200:10:3500 linspace(0,20000,101)]); % RPM

% RotatingLoad NodeId f0 theta0 exponent : define complex rotating load
if s==1 % unbalanced mass
 model=fe_rotor(sprintf('rotatingload 2 %.15g -90 2',mb*db),model);
else % asynchronousload
 model=fe_rotor(sprintf('rotatingload 2 %.15g -90 0',f0),model);
end
r1=struct('Omega',om/60*2*pi,'w',s*om/60*2*pi); % Range
model=stack_set(model,'info','Range',r1);
R1=fe_rotor('forcedresponse',model); % compute forced response
iiplot(R1) % plot response
% Post (radial deformation):
Q=max(abs(R1.Y),[],2); figure;semilogy(om,Q);
xlabel('Rotation speed [RPM]'); ylabel('radial def amplitude [m]')
if s==1; title('Unbalanced mass')
else;    title(sprintf('Asynchronous load %.15g\\Omega',s))
end
```

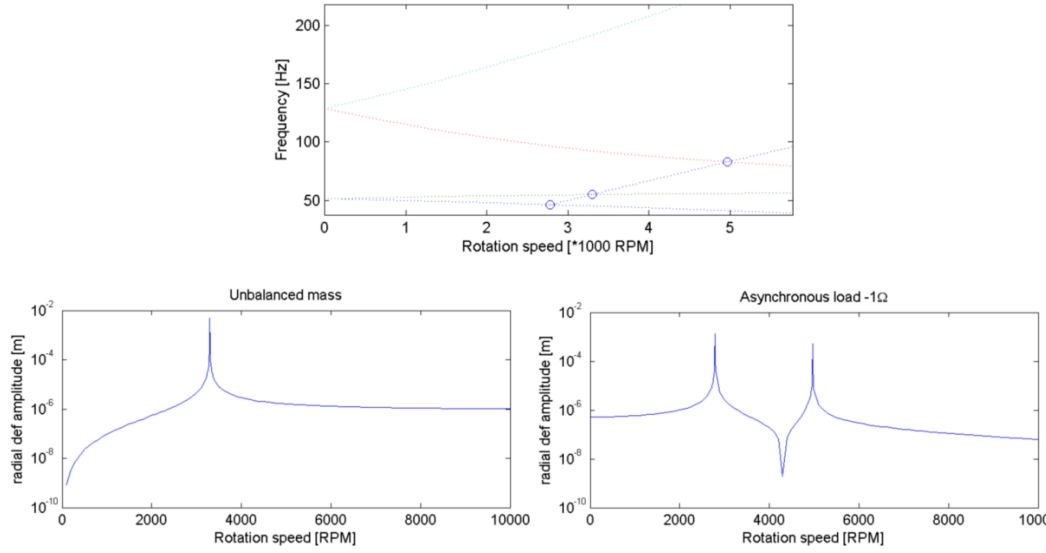

Figure 4.10: Top: Campbell diagrams. Bottom: Responses to unbalanced mass and asynchronous load.

Fig 4.10 shows the radial deformation response for an unbalanced load and for an asynchronous load rotating at $-\Omega$ speed. The unbalanced load excites the forward modes (3296 RPM) whereas the asynchronous load excites the backward modes (2785 RPM and 4697 RPM). Frequencies match those computed as critical frequencies in the Campbell diagram.

4.3.2 1D models in a rotating (body-fixed) frame

While this representation is not very classical, it corresponds to the nominal choice when doing time integration of a rotor that is not axisymmetric.

4.4 3D rotor

The same rotor as described in lalanne (see fig 4.3) is meshed using `hexa8` elements. Use `model=d_rotor('TestVolShaftDiskMdl')`.

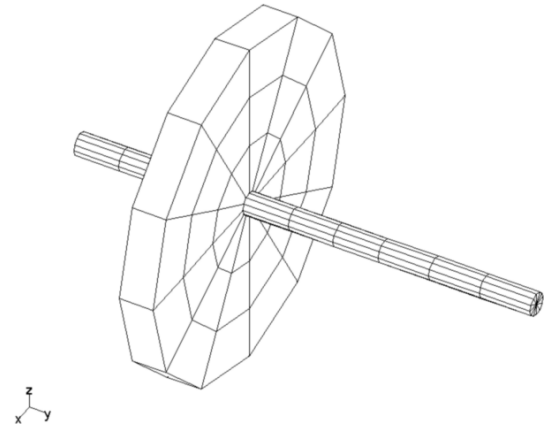

Figure 4.11: 3d model of Lalanne rotor.

Matrices are defined in the local rotating frame. We described the unbalanced load by a static load, and we use in following example the same procedure as for local frame 1d rotor, at $w = 0$.

```
model=d_rotor('testvolshaftdiskmdl')
% Assemble nominal matrices:
model.DOF=[];model=fe_caseg('assemble -se -matdes 2 1 7 8',model);
model.DOF=fe_case('gettdof',model);
% Campbell diagram:
model=stack_set(model,'info','eigopt',[5 20 1e3]);
fe_rotor('campbell -critical',model,linspace(0,20000,30));

% Unbalanced mass or asynchronous load :
mb=1e-4; db=0.15; % mass, distance to axis
s=0; f0=1; % s=1, unbalanced load. s<>1, asynchronous load
om=sort([2789 2750:10:2820 11760:10:11840 linspace(0,20000,101)]); % RPM


 model=fe_rotor(sprintf('rotatingload 180 %.15g 0 2',f0),model);

 r1=struct('Omega',om/60*2*pi,'w',s*om/60*2*pi); % Range
 model=stack_set(model,'info','Range',r1);
 R1=fe_rotor('forcedresponse',model); % compute forced response
 iiplot(R1) % plot response
```

```
% Post (radial deformation):
Q=abs(R1.Y(:,1)); % unbalanced along x
figure;semilogy(om,Q);
xlabel('Rotation speed [RPM]'); ylabel('radial def amplitude [m]')
if s==0; title('Unbalanced mass')
else;    title(sprintf('Asynchronous load %.15g\\Omega',s))
end
```

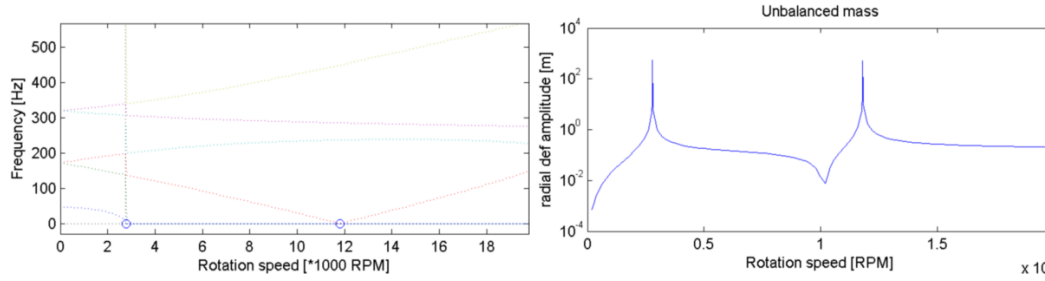

Figure 4.12: Left: Campbell diagram. Right: Response to unbalanced mass.

Unbalanced mass excites the forward whirl modes. Maximum response is found at critical speeds (rotation speeds that induce a complex mode of 0 Hz frequency in the rotating frame). Campbell critical speed (2789 RPM) matches computed frequency response.

4.5 Data structure reference

xxx

Function reference

Contents

fe_cyclic 62
fe_rotor 65
rotor1d 71
rotor2d 74
demo_cyclic 75
fe_cyclicb Mesh 79
fe_cyclicb 83
obsolete 95
nl_spring 103
mkl_utils 112
Non linearities list 114
Creating a new non linearity: nl_fun.m 130
nl_solve 132
nl_mesh 135
spfmex_utils 142
extrotor 144

fe_cyclic

Purpose Support for cyclic symmetry computations.

Syntax

```
model=fe_cyclic('build NSEC',model,LeftNodeSelect)
def=fe_cyclic('eig NDIAM',model,EigOpt)
```

Description fe_cyclic groups all commands needed to compute responses assuming cyclic symmetry. For more details on the associated theory you can refer to [8].

`Assemble [,-struct]`

This command supports the computations linked to the assembly of gyroscopic coupling, gyroscopic stiffness and tangent stiffness in geometrically non-linear elasticity. The input arguments are the model and the rotation vector (in rad/s)

```
model=demosdt('demo sector all');
[K,model,Case]=fe_case('assemble -matdes 2 1 NoT -cell',model);
SE=fe_cyclic('assemble -struct',model,[0 0 1000]); %

def=fe_eig({K{1:2},Case.T,model.DOF},[6 20 0]);% Non rotating modes

def2=fe_eig({K{1},SE.K{4},Case.T,model.DOF},[6 20 0]); % Rotating mode shapes

[def.data def2.data]
```

Note that the rotation speed can also be specified using a stack entry `model=stack_set(model 'info', 'Omega',[0 0 1000])`.

`Build ...`

`model=fe_cyclic('build nsec epsl len',model,'LeftNodeSelect')` adds a cyclic symmetry entry in the model case. It automatically rotates the nodes selected with `LeftNodeSelect` by 2π/nsec and finds the corresponding nodes on the other sector face. The default for `LeftNodeSelect` is `'GroupAll'` which selects all nodes.

The alternate command `model=fe_cyclic('build nsec epsl len -intersect',model,'I` is much faster but does not implement strict node tolerancing and may thus need an adjustement of `epsl` to higher values.

Command options are

- *nsec* is the optional number of sectors. An automatic determination of the number of angular sectors is implemented from the angle between the left and right interface nodes with the minimum radius. This guess may fail in some situtations so that the argument may be necessary.
- nsec=-1 is used for periodic structures and you should then provide the translation step. For periodic solutions, model=fe_cyclic('build -1 *tx ty tz* epsl *len* -intersect',model,'LeftNodeSelect') specifies 3 components for the spatial periodicity.
- Fix will adjust node positions to make the left and right nodes sets match exactly.
- epsl*len* gives the tolerance for edge node matching.
- -equal can be used to build a simple periodicity condition for use outside of fe_cyclic. This option is not relevant for cyclic symmetry.
- -ByMat is used to allow matching by MatId which allows for proper matching of coincident nodes.

```
model=demosdt('demo sector 5');
cf.model=fe_cyclic('build epsl 1e-6',model);
```

LoadCentrifugal

The command is used to build centrifugal loads based on an info,Omega stack entry in the form

```
data=struct('data',[0 0 1000],'unit','RPM');
model=stack_set(model,'info','Omega',data);
model=fe_cyclic('LoadCentrifugal',model);
```

Eig

def=fe_cyclic('eig *ndiam*',model,EigOpt) computes *ndiam* diameter modes using the cyclic symmetry assumption. For *ndiam*¿0 these modes are complex to account for the inter-sector phase shifts. EigOpt are standard options passed to fe_eig.

This example computes the two diameter modes of a three bladed disk also used in the d_cms2 demo.

```
model=demosdt('demo sector');
model=fe_cyclic('build 3',model,'groupall');
fe_case(model,'info')
def=fe_cyclic('eig 2',model,[6 20 0 11]);
fe_cyclic('display 3',model,def)
```

The basic functionality of this command is significantly extended in `fe_cyclicb ShaftEig` that is part of the SDT/Rotor toolbox.

See also Section **??**

fe_rotor

Purpose

The fe_rotor function implements classical solutions for rotor dynamics applications.

AddMass

```
mdl=fe_rotor('AddMass',mdl,RO);
```
This command can be used to add a local mass on a 3d rotor mesh. The mass is added, and linked to existing node using a MPC connection.

RO is a data structure with fields

- .mxyz matrix whose first 3 columns define x y and z coordinates of added mass, and the 4th defines the mass.
- .ProId the ProId of the rotor where mass is added.

Command options are

- -DofLoad defines a 6 direction load on added masses (usefull for reduction purpose).
- -mpcmaster modifies MPC connection so that mass DOF are master ratherthan slave.

Campbell

fe_rotor('Campbell',model,RunOpt) computes the Campbell diagram and displays it with fe_rotor('plotcampbell',out,RunOpt) if no output argument is requested.
RunOpt can be a vector of rotation speeds (RPM) or a data structure with at least field .Omega containing rotation speeds and other fields giving other variables used in zCoef (for example .par field).

Accepted command options are

- `-cf i` can be used to force display of the diagram in a specific figure.
- `-nodir` avoids the call to determine the rotation direction.
- `-full` forces the use of full matrices. Without the argument, full matrix eigenvalue calls are only performed with less than 100 DOFs.
- `-crit` overlays the critical speed computation.
- `-stability` displays stability diagram (damping/Omega).

Examples can be found in section 3.4.1.

Critical

Computation of critical speeds. It can be called when computed Campbell using command option `-crit` (see `Campbell`).
Critical speed are computed assuming deformation is a complex mode at the same frequency as the rotation speed ($w = \Omega$) in the global fixed frame or equal to 0 ($w = 0$) in the local rotating frame.
In the global fixed frame:
$M\{\ddot{X}\} + (C + \Omega D(\Omega = 1))\{\dot{X}\} + (K + \Omega^2 Kc(\Omega = 1))\{X\} = 0$
With $X = X_0 exp(\imath\Omega t)$
$(\Omega^2(-M + \imath D(\Omega = 1)) + \imath\Omega C + K)\{X_0\} = 0$
In the local rotating frame:
$M\{\ddot{X}\} + (C + \Omega D(\Omega = 1))\{\dot{X}\} + (K + \Omega^2 Kc(\Omega = 1))\{X\} = 0$
With $X = X_0$ constant
$(K + \Omega^2 Kc(\Omega = 1))\{X_0\} = 0$

Examples can be found in section 3.4 xxx.

Whirldir

Internal command to display the direction of the modes on the Campbell diagrams. xxx details

Assemble

`model=fe_rotor('Assemble',model,zCoef,r1);` This command is used to assemble mass stiffness and damping matrices taking in account all the rotor matrices (gyroscopic coupling, centrifugal softening, etc. ...).
Default `zCoef` can be obtain using `model=fe_def('zCoef-default',model);. r1`

is a data structure whose fields are parameters used in zCoef. In particular one should specify the rotation speed r1.Omega (rad/s). Command option -cell can be used to return only matrices in a cell array.

RotatingLoad

This command builds a rotating DofLoad on a model node for frequency analysis.
model=fe_rotor('RotatingLoad NodeId f0 theta0 wexponent',model);
NodeId is the id of the node where load will be applied, f0 is the amplitude of the load, theta0 is the angle formed by the load initial direction and the first global direction in the plane orthogonal to rotation axis (x for y and z rotation and y for x rotation). The amplitude of the load can depend on a power of pulsation defined with wexponent.
Resulting complex load is of the form

$$f0w^{\text{wexponent}} \left\{ \begin{array}{c} 1 \\ \pm\imath \end{array} \right\}$$

in the plane orthogonal to the rotation axis (sign of $+/-\imath$ depends on rotation axis).
Forced response to this load can be computed using ForcedResponse.

ForcedResponse

This command can be used to compute forced response to a frequency load (for example a rotating load built using RotatingLoad).
R1=fe_rotor('ForcedResponse',model);
Complex load must be prior defined as a 'In' entry in the case of the model.
Observation can be defined as 'output' SensDof entry. If not, only dof corresponding to the 'In' load will be returned.
Range of computation must be defined in the 'info' 'Range' Stack entry of the model as a structure of data (with fields .Omega defining rotation speeds and .w defining pulsations of the load at corresponding rotation speeds. These fields must be of the same length. If necessary one can add other fields for variables used in a defined zCoef). If .w is equal to zero, frequency dependence of load is assumed to concern the .Omega vector (it is usefull to described unbalanced mass in the local rotating frame since load amplitude depends on Ω^2 but load is static in this frame so that $w = 0$).

Following example defines an unbalanced load on the simple 1d ShaftDisk model defined in d_rotor and computes the forced response (local frame):

```
model=d_rotor('TestShaftDiskMdl') %  build simple model
model=fe_caseg('assemble -se -matdes 2 1 70',model); % assemble model
mb=1e-4; db=0.15; % mass, distance to axis
om=linspace(0,6000,201); % RPM
model=fe_rotor(sprintf('RotatingLoad 2 %.15g -90 2',mb*db),model); % unbalance
r1=struct('Omega',om/60*2*pi,'w',om/60*2*pi); % Range
model=stack_set(model,'info','Range',r1);
R1=fe_rotor('forcedresponse',model); % compute forced response
iiplot(R1) % plot response
```

In global rotating frame, load is static ($w = 0$). Following example deals with a rigid disk in global rotating frame:

```
R1=.01;R2=.15;h=0.03;
md=d_rotor(sprintf('TestVolDisk -dim%.15g %.15g %.15g 2 16',R1,R2,h));
md.DOF=[]; md=fe_caseg('assemble -se -matdes 2 1 7 8',md);
% disk assumed to be rigid :
rb=feutil('geomrb',md); cf=feplot(md); cf.def=rb;
md.K=feutil('tkt',rb.def(:,[1 3 4 6]),md.K);
md.DOF=[1.01 1.03 1.04 1.06]';
md.K{2}=0*md.K{2}; % rigid disk
md.K{2}(1,1)=5e5;md.K{2}(2,2)=5e5; % bearing
md.K{2}(3,3)=1e5;md.K{2}(4,4)=1e5; % bearing rot
% Unbalanced mass or asynchrone load :
mb=1e-4; db=0.15; % mass, distance to axis
s=0;
om=sort([2789 2750:10:2820 11760:10:11840 linspace(0,20000,101)]); % RPM
md=fe_rotor(sprintf('RotatingLoad 1 %.15g 0 2',mb*db),md);
r1=struct('Omega',om/60*2*pi,'w',s*om/60*2*pi); % Range
md=stack_set(md,'info','Range',r1);
R1=fe_rotor('forcedresponse',md); % compute forced response
iiplot(R1) % plot response
```

SEBuild

This command can be used to build a superelement of rotor from a 3d rotor model. It includes Craig Bampton reduction, bearing rings adding, ...

```
SE=fe_rotor('SEBuild',model,RO);
```

`model` is a 3d mesh of rotor.
`RO` is a data structure with following fields

`.xbea` vector defining position of bearings according to rotor axis.

`.bea` data structure of bearing infos. `bea.length` is the length of the bearings, `bea.Npt` the number of points on one bearing, `bea.ProId` is the ProId of the journal.

`.mxyz` is optional. Defines the mass position (reduction is performed so that DOF are created at these positions).

`.EigOpt` eig option for mode computation for Craig-Bampton reduction.

One can use following options:

`-noreduce` no reduction is performed.

`-rigid` use rigid ring for bearings instead of rbe3 rings.

```
% Lalanne 3D rotor
model=d_rotor('TestVolShaftDiskMdl'); % Build model
% remove bearing and boundary conitions:
model.Elt=feutil('RemoveElt proid1000:1001',model); % remove linbearings
model=fe_case(model,'Remove','Ends');
 % Build SE:
RO=struct('mxyz',[0.0894892917244 0.118333333333 0.0516666666667],... Po
          'EigOpt',[5 20 1e3]); % eig option for CraigBampton
RO.xbea=[0;0.4];
RO.bea.length=0.01;
RO.bea.Npt=1;
RO.bea.ProId=1; % proid of the bearing
SE=fe_rotor('SEBuild-rigid',model,RO); % Following is now in SEbuild com
```

`TimeOpt`

```
model=fe_rotor('TimeOpt',model);
```

Defines a default `TimeOpt` in the `'info' 'TimeOpt'` model stack entry for time integration with rotor models. Following command options are available:

- `Assemble` Defines `AssembleCall` field in order to take in account gyroscopic effects in time computation. `TimeOpt` field `.matdes` should be filled by user in order to define what matrices will be assembled. Default is `[2 1 7]` for mass stiffness and gyroscopic coupling matrix, see `sdtweb('MatType')` for available matrices (for example `8` for centrifugal softening...). Before time integration desired matrices will be assembled taking in account the `'info' 'Omega'` model stack entry that defines rotation vector (norm should be equal to one so that matrices can be assembled once for different computation with different rotation speed). Rotation speed is taken as first parameter on the `'info' 'Range'` model stack entry (in that case rotation vector norm must be one). If there is no Range entry, rotation is taken equal to the norm of rotation vector. Corresponding M C K matrices are then built (`fe_rotor Assemble` call) and will be used for Newmark time integration.

rotor1d

Purpose This function gives some commands to help the meshing of 1D rotor. See section 3.1.1 for general details about 1d rotors and section 3.1.2 for a full example of 1d rotor meshing.

`1To2D`

From a 1d model with `beam1` elements (masses are not converted for the moment), builds a 2d rotor.
`LC l` defines the max length of elements.

`1To3D`

From a 1d model with `beam1` elements (masses are not converted for the moment), builds a 3d rotor. one can see section 3.1.3 for a tutorial.

- `LC l` defines the max length of elements.
- `div n` defines the number of sector of 3d mesh.

The grounded celas are assumed to be bearings. RBE3 rings are defined at each bearing.

`AddBearing`

This command add a rotor bearing as a celas element.
`mo1d=rotor1d('AddBearing ...',mo1d,pos);`
`mo1d` is a 1d rotor model `pos` is the x y z position of the bearing.

Command options are

- `-keep` if not present remove existing celas at given position.
- `DOF DOF` that defines the DOF concerned by bearing link.
- `k k` the spring stiffness.
- `c c` the spring damping.

- ProID *ProID* the spring ProID.

For example:

```
mo1d=rotor1d('AddBearing DOF -123 k 1e4 -keep',mo1d,pos);
```

AddBeam

```
mo1d=rotor1d('AddBeam ...',mo1d);
```

This command adds a beam on the axis of the rotor, accordin to following argument specified as command option:

- x1 *x1* beginning of the beam in the axis direction
- x2 *x2* end of the beam in the axis direction
- r1 *r1* inner radius of the beam. By default 0.
- r2 *r2* outer radius of the beam.
- MatID *MatID* MatID. If not given new MatID is used.
- ProID *ProID* ProID. Note that the associated element property contains the section information according to r1 and r2, so that existing ProID will be lost. If not given new ProID is used.
- -refine Refine existing beams to ensure that beams have coincident nodes.

For example

```
mo1d=rotor1d('AddBeam x1 0.5 x2 0.7 r1 0.11 r2 0.77 MatID 1',mo1d);
```

AddNodeRefine

```
[mo1d,ind]=rotor1d('AddNodeRefine ...',mo1d,xyz);
```

This low level call command adds nodes at given positions `xyz` (one row per node, 3 columns x y and z, or one column to define distance along the rotor axis) refining beams of the model `mo1d` if needed.

Command options are:

- -epsl *epsl* Tolerance for new node adding.

For example

```
[mo1d,ind]=rotor1d('AddNodeRefine',mo1d,[0.2 0 0]);
```

Skyline

```
mdl=rotor1d('Skyline...',xy);
```

This command can be used to generate rotor model from skyline description. `xy` is a 2 column matrix whose 1st column defines position according to rotation axis, and 2nd column defines radius at corresponding positions. One can define multiple parts rotor using separator NaN in the 1st column. For 2D rotor meshing, use `[NaN r]` to define internal radius `r` of the corresponding part.

- `SkylineToBeam` builds 1d beam rotor model. See example in 1D subsection of section 3.1.1.

- `SkylineTo2D` a 2D rotor model (that can be used to generate 3D model by revolution, or using cyclic symetry, see `rotor2d BuildFrom2D`)), see example in 2D subsection of section 3.1.1.

Following command options are accepted

- `-Lc l` specify maximum length `l` of elements.

rotor2d

Purpose

The `rotor2d` function lets you build a superelement representation for rotor applications starting from a 2D model. A tutorial is given in section **??**

`teig`

Needs description.

`test`

Simpledisk

`BuildFrom2D`

`model=rotor2d('buildFrom2D',model2D);` builds a 3D sector model by revolution of a 2D section. In the rotor module, the symmetry axis is always z so that if the 2D mesh is given in xy coordinates a permutation is performed.

- `-nsec` *N* number of sectors. By default, one considers 10 sectors, for a different value use `buildFrom2D -nsec` *N*.
- `-div` *N* number of divisions in each sector.

The sector is modeled as a superelement called `disk1`.

```
% Model Initialization
model2D=rotor2d('test simpledisk -back');
cf=feplot(rotor2d('buildFrom2D',model2D));
SE=cf.Stack{'disk1'}; % enforce boundary cond. on sector and assemble
SE=fe_case(SE,'FixDof','Base','z==1.01');
SE=stack_set(SE,'info','Omega',[0 0 0;0 0 1000]); % define speed range
SE=fe_cyclic('assemble -se',SE);
cf.Stack{'disk1'}=SE; fecom('view1');

cf.Stack{'info','EigOpt'}=[5 20 0]; % define eigenvalue options
RunOpt=struct('targ',1, ...      % define target diameter
   'Range',linspace(0,1,30));    % define speed points relative to range
[cf.def,hist]=rotor2d('teig',cf,RunOpt);
figure(1);rotor2d('plot',hist);set(gca,'ylim',[0 250])
```

demo_cyclic

Purpose Combines examples of the use of `fe_cyclicb` commands.

Syntax

```
demo_cyclic('testdisk nsec')
demo_cyclic('testrotor nsec1 nsec2 ...')
```

Disk

Moved to section 3.5.2.

ShaftMono

Moved to section 3.5.3.

Variable speed

One considers stiffness matrices that are dependent on the rotation speep. Assuming that a second order polynomial representation is sufficient, one can build a vector of weigthing coefficients

$$\left\{ \begin{array}{c} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{array} \right\} = \left[\begin{array}{ccc} 1 & 1 & 1 \\ \Omega_1 & \Omega_2 & \Omega_3 \\ \Omega_1^2 & \Omega_2^2 & \Omega_3^2 \end{array} \right]^{-1} \left\{ \begin{array}{c} 1 \\ \alpha_2 \\ \alpha_3 \end{array} \right\} \tag{5.1}$$

Such that the stiffness at rotation speed Ω is approximated by

$$[K(\Omega)] = \sum_{\alpha_i} [K\Omega_i] \tag{5.2}$$

The `zCoef` uses velocity `Omega` in rad/s.

This example now treats computation at variable rotation speeds

```
% Model Initialization
model=demo_cyclic('testblade');cf=feplot(model);
% Compute matrix coefficients for a multi-stage rotor
range=struct('data',[0 0 1;0 0 800;0 0 1600],'unit','RPM');
% Assembling in the feplot figure, allows memory offload
fe_cyclicb('polyassemble -noT',cf,range);
```

```
X=struct('data',linspace(0,1600,10),'unit','RPM');
fe_rotor('Campbell -cf1',model,X)
```

Another example will be needed to treat the multi-stage case

```
% Model Initialization
model=demo_cyclic('testrotor 7 10 -blade');
model=fe_cyclicb('shaftRimAsSe',model);
cf=feplot(model);

% Compute matrix coefficients for a multi-stage rotor
range=struct('data',[0 0 1;0 0 800;0 0 1600],'unit','RPM');
% Assembling in the feplot figure, allows memory offload
fe_cyclicb('polyassemble -noT',cf,range);

% Now run a mono-harmonic multi-speed computation
cf.Stack{'info','Omega'}=struct('data',range.data(1,:),'unit','RPM');
def=fe_cyclicb('shafteig 0 -ReAssemble 2 -NoN',cf);
Sel={'disk1','groupall';'disk2','groupall'};
fe_cyclicb('DisplaySel',cf,def,Sel)

% Reduce the full model
fe_cyclicb('ShaftPrep -svdtruncate -mseq1 -handle -norestit',cf,def);
fesuper('fassemble',cf);

% Force single harmonic
% xxx
```

Rewrite needed here.

ShaftMulti

The second example is a non-monoharmonic shaft computation.

The following example builds a reduced order model from a set of mono-harmonic modeshapes whose Fourier harmonics are 0, 1 and 2 and sector modes with fixed interfaces. The latter are computed within the framework of mono-harmonic computations, they are called with -1 in shaftTeig. No confusion is possible since the

true mono-harmonic solutions with $\delta = -1$ are solutions with $\delta = 1$. However, the restitution of fixed interface solutions with `Display[Sel]` has no sense.

Call `shaftprep` aims to build the reduced kinematic subspaces of the sector super-elements from the specified target solutions. Prior to that, it separates the sector mesh into two: the slice with the left-interface nodes (the inter-sector super-element) and the remaining elements (the sector super-element). It projects the matrices of the sector super-elements onto their individual subspaces and the matrices of the inter-sector slices onto the subspaces of its two neighbouring sector super-elements.

Command `fassemble` in `fesuper` first projects the finite elements matrices of the inter-disk volumic interface onto the subspaces of its neighbouring disks. It then assembles the reduced matrices of the sector super-elements and inter-sector slices to form the reduced matrices of the disks. Finally, it assembles these reduced matrices and that of the volumic interfaces to form the reduced matrices of the whole rotor.

Here too, a selection can be specified so that the generalized modeshapes can be recovered to a subset of physical DOF (relying on the true mesh or a viewing mesh). The selections are defined like in `DisplaySel`, however both the sector and inter-sector super-elements have to be considered so that the recovery concerns the whole underlying bladed sector.

The Fourier harmonic contents of the generalized modeshapes can be obtained without recovery with the help from command `fourier` of `fe_cyclicb`. When specified, `-egyfrac` returns the fraction of strain energy per harmonic per disk so that energy localization within a disk can be achieved with the supplementary information of which harmonics are involved in the response. The graph displays the disks (from top to bottom) and for each disk the possible harmonics between 0 and $N/2$ (if applicable). Another way to display the same information is to group the harmonics first and then the disks. This is done with the `-sortbyd` option. It adds the proper amount of zeros for harmonics that are not present on a given disk.

`_mono.mat` to name a model that contains the mono-harmonic description.

```
% Model Initialization
cf=demo_cyclic('testrotor 7 10 -blade -cf 2');

% Mono-harmonic Solutions
model=stack_set(cf.mdl,'info','EigOpt',[5 10 -1e3 11 1e-8]);
[def,sectors]=fe_cyclicb('shaftteig -1 0 1 2',model); %-batch
Curve=fe_cyclicb('fourier 1:50 -mono -egyfrac',cf,...
           fe_def('subdef',def,def.data(:,2)~=-1));
```

```
% xxxNeed : display curve fe_cyclicb('fourier -mono -egyfrac -cf 3',Curve);
r1=sortrows(def.data);
if any(r1(1:6,1)>r1(7,1)/1e5);error('Missing rigid modes');end

% Model Reduction
fe_cyclicb('ShaftPrep -svdtruncate -mseq1 -handle -norestit',cf,def);
fesuper('fassemble',cf);

% Mode Computations
defr=fe_eig(cf.Stack{'MVR'},[5 50 1e3 11 1e-8]);

Sel={'disk2','innode {r>150}';
     'disk2l','innode {r>150}';
     'disk1','groupall';
     'disk1l','groupall';
     '','withnode {x>0}'};
fesuper('sebuildsel -initrot',cf,Sel);
cf.def=feutilb('placeindof',cf.sel.cna{1}.adof,defr);
fecom('colordataevala');

% Post-processing: spatial spectra
cf.sel='reset';cf.def=[];
fe_cyclicb('fourier 7:25 -red -egyfrac -cf 4',cf,defr);
fe_cyclicb('fourier 7:25 -red -egyfrac -sortbyd -cf 5',cf,defr);

% focus on mode 7
fe_cyclicb('fourier 7 -red -egyfrac -cf 6',cf,defr);
```

Now one wants to treat the case of a forced response for loads defined on the disks only. The command `ShaftLoadMulti` allows to build one mono-harmonic excitation per disk. For each disk i, the shape `def_`i and harmonic coefficient `delta_`i are specified in a cell array whose a typical line is {'disk$i$',def_i,delta_i}.

Call `fourier` provides a means to checkout the spatial harmonic content of the generalized load with respect to the set of generalized coordinates (option `-red` has to be specified). Energy-based computations are not available for generalized loads.

Purpose

MeshAddRim

MeshAddRim deals with already existing inter-disk volumic rings. It renumbers their model to be integrated with the already defined disk super-elements. Note that adjusting the geometric tolerance with option epsl val can be important.

```
cf=demo_cyclic('buildstep0');% See sdtweb('demo_cyclic.m#Step0')

% Extract rim model into mo1 (would come from other reading)
[cf.mdl.Elt,mo1.Elt]=feutil('RemoveElt EltName~=SE',cf.mdl);
mo1.Node=cf.mdl.Node;mo1.Node=feutil('GetnodeGroupAll',mo1);
cf.mdl.Node=[];

% Combine mo1 (rim) and cf.mdl (disks)
fe_cyclicb('MeshAddRim epsl 1e-3',cf,mo1);
cf.sel='EltName~=SE';fecom('showpatch');
```

Other commands documented in the tutorial are

- MeshRim described below and illustrated in section 3.1.6, is an automated procedure to mesh rims form a selection of nodes on two disks.
- MeshRimLine2Patch, described in section 3.1.6 is used to build view meshes of large models.

MeshRim

This command is used to mesh rims as volumes or penalty springs.

- -kp val is used to specify the penalty stiffness.
- epsl val specifies length tolerance for node matching.
- -masterdisk val specifies the disk(s) on whose surface the slave nodes will be matched.
- -masterdisk val specifies the slave disk(s) whose edge nodes will be connected to the master disk surface.

MeshSecAddNode

MeshSecAddNode [epsl *espl*] is used to place sensors relatively to a disk/rotor model. It uses a cell array to define sensors (see sdtweb('sensor#scell')). It first matches each sensor to a surface in a sector of a disk with an optional tolerance *epsl* that allows to tune the research. It then adds these surface elements to the selection. The latter serves as the input of the subsequent SensMatch call that

1. automatically deals with the specified direction of measure (column 'DirSpec') by detecting the normals, computing vectors from the latter, *etc.*,
2. builds the observation matrix at each sensor (cta entries) that relates the DOF of the matched surface to the observation at this sensor along the measurement direction.

```
% Load 360 reduced shaft model, sdtweb('demo_cyclic.m#Step2')
cf=demo_cyclic('buildstep2');

sensors=cf.Stack('Test').tdof;
disp(sensors); % display sensor properties

% match sensors
[sens,sel]=fe_cyclicb('MeshSecAddNode epsl 20',cf,sensors);
cf.mdl=fe_case(cf.mdl,'SensDof','Test',sens);

% display sensors
fecom('showpatch');
fecom(cf,'curtab Cases','Test');
```

The model to which the sensors are matched is made of super-elements already reduced, call to MeshSecAddNode with the supplementary input of a generalized quantity (with a list of generalized coordinates stored in .DOF) automatically builds the observation matrix that expands the generalized modeshapes to the sensors along the measurement direction.

A first application of this is to animate generalized modeshapes on the experimental wire frame as proposed in the example below.

```
cf=demo_cyclic('buildstep4');
def=cf.Stack('def_mvr');
```

```
% match sensors
sensors=stack_get(cf.mdl,'info','Test','GetData');
[r1,sel,sens]=fe_cyclicb('MeshSecAddNode epsl 20',cf,sensors.tdof,def);
% add wireframe
r1.Elt=[r1.Elt zeros(size(r1.Elt,1),1);cf.Stack('WireFrame')];
r1.Elt=feutil('setgroupall egid -1',r1);
cf.mdl=fe_case(cf.mdl,'SensDof','Test',r1);

% animate generalized modes at sensors
cf.sel(1)='groupall';
cf.sel(2)='-Test';
cf.def=struct('def',sens.cta*def.def,...
              'DOF',cf.CStack('Test').tdof(:,1),...
              'data',def.data);
cf.o(1)={'sel 2 def 1 ch 1 ty8 scc10','edgecolor','r','linewidth',2};
```

A second application is to display and animate generalized response or transfer functions.

```
cf=demo_cyclic('buildstep6');
xF=cf.Stack('xF_mvr');

% match sensors
sensors=stack_get(cf.mdl,'info','Test','GetData');
[r1,sel,sens]=fe_cyclicb('MeshSecAddNode epsl 20',cf,sensors.tdof,xF);
% add wireframe
r1.Elt=[r1.Elt zeros(size(r1.Elt,1),1);cf.Stack('WireFrame')];
r1.Elt=feutil('setgroupall egid -1',r1);
cf.mdl=fe_case(cf.mdl,'SensDof','Test',r1);

% display responses
xF=cf.Stack('xF_mvr');
ci=iiplot(3);iicom(ci,'curvereset');
XF=iicom(ci,'curveXF');
XF('Test').w=xF.data';
XF('Test').xf=(sens.cta*xF.def)';
XF('Test').lab=sens.lab;
iicom('SubMagPha');
iicom(ci,'IIxOnly',{'Test'});

% animate response at sensors
```

```
cf.sel(1)='groupall';
cf.sel(2)='-Test';
cf.def=struct('def',sens.cta*xF.def,...
              'DOF',cf.CStack('Test').tdof(:,1),...
              'data',xF.data);
cf.o(1)={'sel 2 def 1 ch 1 ty8 scc10','edgecolor','r','linewidth',2};
```

MeshSensMatch

supports topology correlation for a SensDof entry defined on a multi-stage shaft model.

MeshSurfSet

generates standard surface sets for a sector.

MeshCylSurf

model=fe_cyclicb('MeshCylSurf epsl *val*',model,NodeList) makes a surface perfectly cylindrical. The node list of nodes to be used on the left edge may be omitted (it will be found automatically).

MeshFixTheta

provides handling utilities for slanted sectors, see section 3.1.5.

MeshFixRadial

Generates MPC constraints for using DOFs given in radial coordinates: .01 is radial, .02 tangential, .03 axial.

The following example illustrates a case where tangential motion of nodes 5 and 6 is set.

```
cf=demo_cyclic(sprintf('testrotor %i -cf 5',5));
cf.Stack{'disk1'}=fe_cyclicb('MeshFixRadial', ...
   cf.Stack{'disk1'},'radial',[5.02;6.02]);
def=fe_cyclicb('shaftteig 0 1', ...
 stack_set(cf.mdl.GetData,'info','EigOpt',[5 3 0]))
fe_cyclicb('Display',cf,def);
fecom('ColorDataEvalTanZ -ColorBarTitle"TanZ"');
```

fe_cyclicb

Purpose Support for advanced cyclic symmetry computations.

Description fe_cyclicb groups advanced commands used to build and manipulate reduced order models of single symmetric structures and their assemblies. For more details on the associated theory you can refer to [8].

Rotor Construction
DiskFromSector

This command builds a disk/rotor model from (a) physical sector model(s). Shafts can be generated by combining multiple calls to disk from sector. Once, disks are combine, their connection trough rim models is described in section 3.1.6.

Command DiskFromSector also handles the construction of the cyclic sector models. Cyclic symmetry information can be already given in the sector model (calls to fe_cyclic('build') done beforehand) or done in the command. In the later case, an optional **epsl** *tol* can be declared so that it is propagated to the subsequent call to fe_cyclic('build epsl *tol*',...), where *tol* is the desired tolerance for left-right interface node matching.

The example below demonstrates the capability of the function for two disks with 7 and 10 blades respectively.

```
cf=demo_cyclic('buildstep0');
sector1=cf.Stack{'disk1'};
sector2=cf.Stack{'disk2'};

% build disk1 from sector1
model=fe_cyclicb('DiskFromSector epsl 1e-6',[],sector1);
% build disk2 from sector2 and append to disk1
model=fe_cyclicb('DiskFromSector epsl 1e-6',model,sector2);
fe_cyclicb('DisplayFirst',model) % Avoids full display for large models
```

In cases when cf already contains one sector per disk, the shaft model can be created in a single operation with the command fe_cyclicb('diskfromsector',[],cf,{'s
where **sel_disk***i* selects the sector model of disk i. The example below illustrate this by putting the two sector models into a single one prior to the rotor assembly.

```
cf=demo_cyclic('buildstep0'); % See sdtweb demo_cyclic('Step0')
```

```
sectors=cf.Stack{'disk1'}; % Build a model with two sectors
sectors=feutil('addtest',sectors,cf.Stack{'disk2'});
sectors.Stack={};
cf.model=sectors;

% build rotor from sectors and auto display
fe_cyclicb('DiskFromSector epsl 1e-3',cf,{'group1:2','group3:4'});
```

During the build process, sectors are automatically renumbered so that node numbers are left interface, interior, right interface (in order matching that of the left interface). The renumbering can be forced with the `-renumber` option. This allows to have nodal overlap between the superelements of two adjacent sectors. The command then adds a `mpc,diski_end` multiple point constraint to account for the fact that the disk is closed circumferentially.

Mesh

Meshing utilities See `fe_cyclicb Mesh`.

ConnectionRing

`ConnectionRing` builds a "ring connection" where the structure is fixed axially and radially on a set of nodes and first point only in tangential direction

Display

`Display` commands group tools to build mesh views specific to disk assemblies.

- `[def,ENER]=fe_cyclicb('Displaysel',cf,def,Sel,'enerkdens')` is used to recover mono-harmonic solutions on a partial selection For details on mono-harmonic solutions see section 3.5. Examples can be found in section 3.5.2, section 3.5.3, ...

  `Sel` is a cell array specifying how each stage is displayed. In the example from section 3.5.1, one uses

  ```
  Sel={'','EltName SE';    % Keep only SE for display (no interstage rim)
       'disk1(1:2)','groupall'; % all elements from sectors 1:2
       'disk2(1:3)','groupall'};% all elements from sectors 1:3
  ```

  See `sdtweb fesuper#SeBuildSel` for details on partial superelement display and more examples on the way to define `Sel`). The last command can be any valid `fe_stress` command. Without output argument the result is displayed.

```
[cf,def]=demo_cyclic('buildstep1');
def=fe_def('subdef',def,def.data(:,2)>=0); % remove fixed edge solu
fe_cyclicb('Display',cf,def);fecom('ColorDataEvalA');

fe_cyclicb('DisplaySel',cf,def,cf.Stack('ViewMesh'));
fecom('ColorDataEvalA');
```

During the command one defines `SE.cGL0` corresponding to a rotation by one sector. And the `SE.Alpha` for the harmonic shift. `fesuper('sedefinit -rot',cf);` is then used to define a restitution by disk. The `SeRestit` then contains the `def` so that `SeDefinit` is performed for every restitution.

- `fe_cyclicb('Display',cf,def)` defines a full disk selection within `feplot`. `def=fe_cyclicb('Display',cf,def)` is used to recover full motion from mono-harmonic solutions. For large models, restitution on the full shaft model may be very costly (remember that one vector for 1e6 DOF requires 7.6 MB) and `DisplaySel` is typically preferred.

- `fe_cyclicb('DisplayAllEdges',cf[,sel])` displays 2D cuts of the disks specified in the cell array `sel`, with a typical entry `{'diski'}` to display only disk i (default displays all disks). This cut is basically the projection of the right interface (or equivalently the left interface when meshes are compatible) to the plane $\theta = 0$. Such a view is particularly well suited to the definition of the inter-stage rim nodes in `MeshAddRim` as well as to the construction of viewing meshes in `MeshRimLine2Patch`. Note finally that this view keeps the elements in their original groups.

  ```
  cf=demo_cyclic('buildstep0');
  fe_cyclicb('DisplayAllEdges',cf);
  fecom('colordatag');
  ```

- `fe_cyclicb('DisplayFirst',cf[,sel])` provides a simple command to display the first sector of each stage (`cf` can be replaced by a model resulting from `fe_cyclicb DiskFromSector`). A selection can also be specified to restrict the view to a subset of stages.

  ```
  cf=demo_cyclic('buildstep0');
  fe_cyclicb('DisplayFirst',cf,{'disk2'});
  ```

- `fe_cyclicb('DisplaySkin',cf[,sel])` displays the outer enveloppe of the selected disks in `sel` with the inter-sector common surfaces removed.

```
cf=demo_cyclic('buildstep0');
fe_cyclicb('DisplaySkin',cf,{'disk1'});
fecom('showpatch');set(cf.o(1),'FaceAlpha',.33);
```

- `fe_cyclicb('DisplayInterDisk',cf,nodes)` displays the two ring surfaces used to define the inter-stage volumic interface.

```
cf=demo_cyclic('buildstep0');

% rim nodes
n1=feutil('getnode NodeId',cf.mdl,[12 18 24 1127 1133 1139]');
fe_cyclicb('DisplayInterDisk',cf,n1);
```

- `fe_cyclicb('DisplaySymmetrySurface',cf)` displays the nodes in a cyclic symmetric condition as colored surfaces.

Mono-harmonic solutions

These commands apply to sector models used to compute mono-harmonic eigensolutions.

ShaftTEig,ShaftEig,ShaftSolve

These commands compute mono-harmonic solutions with specified Fourier harmonics (classical cyclic solution for single stage models). For a *tutorial on generating the proper models*, see section 3.1.6. For the associated theory, please refer to [9].

The calling format is `def=fe_cyclicb('shaft Teig delta_list',model);`. `ShatTeig` accepts multiple diameters in the *`delta_list`* and packages individual calls that to `ShatTeig`. For a disk example section 3.5.2.

The main variants in the call are as follows

- Diameter `-1` is used to ask for the computation of fixed edge modes.
- The eigenvalue options `'info','EigOpt'` should be set in the model stack.
- `-ReAssemble` forces reassembly rather than reuse of disk matrices that may have been precomputed and saved in the sector superelements.
- `-thermal` takes thermal loading into account. Thermal state should be stored as a case entry called `DofSet,ThermalState`. See example in section 3.5.1.

- for computations at prestressed state `curve,StaticState` should be defined.
- `-all` xxx
- `-FixTan` is used to enforce no tangential motion of one interface node. This is used for static analysis of freely rotating rotors (example in section 3.5.1).
- `-NoN` stores the imaginary component of the eigenvector using a DOF shift by 50 (thus `.51` is the imaginary x translation). This is necessary if further computations require complex fields (by considering different components there is no difficulty storing the spatial Fourier transform of a complex field).
- `model.Dbfile` when the field is defined to a proper file name, intermediate matrices above the preference `getpref('SDT','OutOfCoreBufferSize',100)` (in MB) are stored in the database file which uses the standard HDF5 based `.mat` format (MATLAB ¿= 7.3).
- `-nlstep` *tol* is used to compute the large non-linear large transformation problem with a tolerance $\delta q/q <$ tol. See an example in section 3.5.1.
- For static computations, the centrifugal load is rebuilt at each step using `model=fe_cyclic('loadcentrifugal',model);` for the rim and sectors.
- `-batch` option is used to compute eigenvectors with multiple diameter in a single run. For large models, this can take many hours so that intermediate file saves are used to allow restarts.

  One then typically expects to have set a `cf.mdl.Dbfile='file_DB.mat'` to allow memory off-loading during the computation.

```
cf=demo_cyclic('testrotor 7 -blade -cf 2');
root=fullfile(sdtdef('tempdir'),'Disk_7_Batch')
%setpref('SdtRuntime','ExecLocal',1) % May be needed
% cf.mdl.Dbfile=[root '_DB.mat']; % out of core matrices

fe_cyclicb('shaft Teig 0 1 5 -batch',[root '.mat']);
% multiple Disk_7_batch_diam.mat files are generated

% Now reload pointers to selected solutions
RO=struct('Fmax',8000,'diams',[0 1 5]);
d1=fe_cyclicb('DefList',root,RO);
```

ShaftSeAssemble

`fe_cyclicb('ShaftSeAssemble -force',cf.mdl,fname)` is used to assemble superelement matrices of each of the `disk*` superelements. If a `curve,StaticState` is defined in the model stack, `fe_cyclic` assemble is used, otherwise `SE=fe_mknl(SE,'NoT')` is called.

If the `-force` option is omitted an attempt to reload the variable `Stack_SE_diski` from the file is first made and assembly is only performed if that variable does not contain the matrices.

If a `mdl.Dbfile` field is defined, the argument `fname` may be omitted.

`-reset` xxx.

Deflist,Def ...

Cyclic symmetry results can be stored in three main forms

- the basic form, valid for real vectors only, stores real and imaginary components for the spatial fourier transform as real and imaginary components at a single DOF
- the long vector form uses additionnal DOFs shifted by 50. Starting from the basic form one would have `d1.DOF=[d1.DOF;d1.DOF+.5]; d1.def=[real(d1.def);imag(`

  The test for usage of this format is that the last dof is above .05 `rem(d2.DOF(end),1)>.5`.
- the double vector form uses the nominal DOFs of the first sector but store the real and imaginary parts as consecutive vectors.

  `d1.def=[real(d1.def(:,1));imag(d1.def(:,1)]; d1.data=[d1.data(1,:);NaN NaN]`

`DefDouble`, `DefLong`, `DefBasic` commands allow transformations between formats while handling out-of-core files properly.

When reading results, `def=fe_cyclicb('DefList','root');` reads all `root*.mat` files and combines the vectors into a single deformation set in the double vector format. Selection of diameters and frequency range during the read process is peformed using

```
RO=struct('Fmax',8000,'diams',[0 1 5 18]);
d1=fe_cyclicb('DefList','root',RO);
```

Specific cases require to sort the output vectors according to the list of diameters specified in the `.diams` field, especially when one wants to put fixed interfaces solutions first for reduction purposes. The command to use becomes `DefListSortDiam`.

Full rotor SE model
`ShaftPrep`

`fe_cyclicb('ShaftPrep',cf,def);` generates reduced sector super-elements.

Each bladed sector is divided into two regions. A first super-element is attached to the elements with the left interface nodes, it is called the inter-sector super-element. A second one is attached to the remaining elements to form the sector super-element.

A reduced kinematic subspace of the sector super-element (with the definition above) is built from `def`, disk by disk.

Vectors are first sorted with respect to their contribution to the considered disk if the `-svdtruncate` option is used.

Then, they are sorted according to their contribution to subsets of physical DOF of the initial sector. If one specifies `-mseq 0` (default call), these subsets are

1. DOFs within either the inter-sector interface (right interface) or the inter-stage interface(s),
2. remaining DOFs (left interface and interior DOF).

If `-mseq 1` is enforced, these subsets are

1. DOFs common to the inter-stage and inter-sector interface(s),
2. DOFs within the inter-sector interface (right interface),
3. DOFs within the inter-stage interfaces,
4. remaining DOFs (left interface and interior DOFs).

Both these sortings make the subsets of vectors linearly independent from each other. They require that fixed edge solutions are stored at the beginning in `def`.

The following step is to make the vectors linearly independent within each set. Vectors in sets (1.), (2.) and (4.), when applicable, are processed with an Iterative

Maximum Sequence Algorithm ([10]). Vectors in set (3.), when applicable, are processed with a Singular Value Decomposition.

See Ref. [5] for details.

-handle option controls whether the resulting bases are stored in memory or on the disk.

-norestit suppresses the explicit construction of the *Restit* variable, normally stored in cf.mdl.Stack.

Once the sector superelements have been generated, the disk model is assembled using the subsequent fesuper('fassemble',cf) call which generates the reduced disk model in cf.Stack{'SE','MVR'}.

A compact example is provided below. A fully developed example can be found in demo_cyclic ShaftMulti.

```
[cf,def]=demo_cyclic('buildstep1');
fe_cyclicb('ShaftPrep -svdtruncate -mseq1 -norestit',cf,def);
fesuper('fassemble',cf);
```

ShaftLoad,ShaftSELoad

fe_cyclicb('ShaftLoadMulti',cf,data); generates a reduced mono-harmonic load for each disk specified in data. It is a cell array whose a typical line is {'disk*i*',def_*i*,delta_*i*} where for each disk i, the shape def_*i* and harmonic coefficient delta_*i* are specified.

fe_cyclicb('ShaftSELoad',cf,def); generates a reduced load from its physical counterpart. In practice, it is used for very specific loading cases, *e.g.* single DOF load or random load.

Developed examples are presented in demo_cyclic ShaftMulti.

```
% define model
cf=demo_cyclic('buildstep4');

% build the excitation on first all sectors, with specified
% diameters 3 on disk1 and 4 on disk 2
data={'disk1',cf.Stack{'disk1'}.Stack{'Enforce_mode_7'},3;
      'disk2',cf.Stack{'disk2'}.Stack{'Enforce_mode_7'},4
      'disk1l',struct('def',1,'DOF',cf.Stack{'disk1l'}.Elt(2,1)+.01),3;
      'disk2l',struct('def',1,'DOF',cf.Stack{'disk2l'}.Elt(2,1)+.01),4};
def34=fe_cyclicb('ShaftLoadMulti',cf,data);
def34.def=sum(def34.def,2); % one column in def34.def per row in data
```

```
def34.data=[0 0];
fe_cyclicb('fourier 1 -red -cf 1',cf,def34); % checkout shape

% now keep the same shapes but force delta=0 on both disks
data(:,3)={0};
def00=fe_cyclicb('ShaftLoadMulti',cf,data);
def00.def=sum(def00.def,2); % one column in def00.def per row in data
def00.data=[0 0];
fe_cyclicb('fourier 1 -red -cf 1',cf,def00); % checkout shape

% now try a random load
r1=fesuper('fnode',cf.mdl);
r1=[r1(:,1)'+.01;r1(:,1)'+.02;r1(:,1)'+.03];
defrnd=struct('DOF',r1(:),'def',[]);
defrnd.def=randn(length(defrnd.DOF),1);
defrnd=fe_cyclicb('ShaftSELoad',cf,defrnd);
defrnd.data=[0 0];
fe_cyclicb('fourier 1 -red -cf 1',cf,defrnd); % checkout shape

% now consider a physical load on first sector of disk 1
r1=fesuper('fnode',cf.mdl);
r1=[r1(:,1)'+.01;r1(:,1)'+.02;r1(:,1)'+.03];
defsp=struct('DOF',r1(:),'def',[]);
defsp.def=fe_c(defsp.DOF,1+[.01;.02;.03])'*randn(3,1);
defsp=fe_cyclicb('ShaftSELoad',cf,defsp);
defsp.data=[0 0];
fe_cyclicb('fourier 1 -red -cf 1',cf,defsp); % retrieve a Dirac's comb
```

ShaftFRF [, D, MS, -rest]

This command allows to build the Frequency Response Functions of a rotor model, either full or reduced. A load and a set of observation DOF have to be defined and added to the model with fe_case. The frequency range is stored in the stack as a 'info','Freq' entry.

The general call is

```
xF=fe_cyclicb('ShaftFRFD',disk,lossfac)
xF=fe_cyclicb('ShaftFRFD -rest',disk,lossfac,cf,sel)
xF=fe_cyclicb('ShaftFRFMS',disk,def,damp)
xF=fe_cyclicb('ShaftFRFMS -rest',disk,def,damp,cf,sel)
```

The command FRFD assembles the matrices of the model then uses them to compute the response. An optional loss factor can be specified.

The command FRFMS synthetizes the response from a set of modeshapes. A damping ratio for all modes can be specified.

The option -rest restores the response computed on the reduced model to a given selection of physical DOF. Without selection, the response is restored to the whole physical DOF set. This option must be disabled when dealing with a full rotor model.

The example developed in demo_cyclic ShaftMulti builds the sythetized response of a reduced rotor model to a random excitation.

```
% define model
[cf,load,def]=demo_cyclic('buildstep5');

% Compute the response to the random excitation
mdl=fe_case(cf.Stack('mvr'),'dofload','Load',load);
mdl=stack_set(mdl,'info','Freq',linspace(900,2000,2201));
xF=fe_cyclicb('shaft frfms',mdl,def,.001);

% select where to restore (upper blade corner)
Sel1={'','eltname SE';'disk1','selface & withnode{NodeId 154}'};
Sel1=fesuper('SeBuildSel -initrot',cf,Sel1);
Sel2={'','eltname SE';'disk2','selface & withnode{NodeId 154}'};
Sel2=fesuper('SeBuildSel -initrot',cf,Sel2);
% ... and do restore
xF1=fesuper('SeDef',Sel1.cna{1},xF);
xF2=fesuper('SeDef',Sel2.cna{1},xF);

% plot responses
ci=iiplot(3);
XF=iicom(ci,'curveXF');
XF('Disk1')=struct('w',xF.data,'xf',xF1.def','dof',xF1.DOF);
XF('Disk2')=struct('w',xF.data,'xf',xF2.def','dof',xF2.DOF);
iicom('subMagPha')
iicom(ci,'IIxOnly',{'Disk1','Disk2'});
ii_plp(def.data);
```

Fourier [,ind], ... [-phys, -mono, -red, -egy, -egyfrac, -sortbyd]

This command allows to perform a 3D Fourier analysis of given modeshapes. The maximum norm of each harmonic is plotted against the harmonic coefficient. The plot is different when dealing with a single modeshape or a set of modeshapes.

Accepted options are

- ind is an optional selection of deformations. See also the alternate fe_def SubDef.
- -phys, -mono and -red are used to distinguish between an analysis of physical modeshapes (full 3D), mono-harmonic modeshapes (the DFT step is omitted) and generalized modeshapes (reduced multi-harmonic model). In all cases, the user has to check that the physical or reduced models are geometrically periodic, *i.e.* that DOF come in repetitive groups, except for -mono where the concept of mono-harmonic modeshapes assumes that structures are periodic.
- -egy and -egyfrac provides means for energy-based computations. Option -egy displays the fraction of energy in each existing harmonic within each disk so that the total amount of energy within each disk is 1. Option -egyfrac displays the fraction of energy in each existing harmonic within each disk so that that total amount of energy within the rotor is 1. The default displays these quantities disk by disk (from top to bottom) and for each disk, all the possible harmonics are displayed (from bottom to top), as depicted in the figure below.
- -sortbyd groups these quantities first by harmonics (from bottom to top) and then by disk (from top to bottom), with the appropriate number of zeros for non present harmonics (typically when $\delta > N/2$ for a given disk), as displayed in the figure below.

```
[cf,def]=demo_cyclic('buildstep1'); % sdtweb demo_cyclic('step1')

Curve=fe_cyclicb('fourier 1:50 -mono -egyfrac',cf,...
           fe_def('subdef',def,def.data(:,2)~=-1));
'xxx'%fe_cyclicb('fourier -mono -egyfrac -cf 3',Curve);

%  sdtweb demo_cyclic('step4') multi-harmonic analysis
[cf,def]=demo_cyclic('buildstep4'); %
fe_cyclicb('fourier 7:25 -red -egyfrac -cf 11',cf,def);
fe_cyclicb('fourier 7:25 -red -egyfrac -sortbyd -cf 13',cf,def);
```

See also Section **??**

obsolete

Purpose Obsolete functionality

Syntax

```
model=fe_cyclicb('Basis [, -norm, -all, -int, -rb]',model,orders,omegas,
def=fe_cyclicb('DiskEig',DISK)
def=fe_cyclicb('DiskEngineLoad EO',model)
def=fe_cyclicb('DiskFRFD [, -rest]',DISK,lossfac,cf,sel)
def=fe_cyclicb('DiskFRFMS [, -rest]',DISK,def,damp,cf,sel)
K=fe_cyclicb('DiskMatrices [ mk]',DISK,Eltselection)
fe_cyclicb('Fourier MODENUM [, -phys, rotor, disk, -red, -test TOL]',fig
model=fe_cyclicb('PolyAssemble [, NoT]',model,params)
DISK=fe_cyclicb('Reduce NODEID0 ELTID0 [, -int]',sector,def)
model=fe_cyclicb('Renumber',model)
def=fe_cyclicb('Display',cf,def)
def=fe_cyclicb('ShaftEngineLoad EO',model)
def=fe_cyclicb('ShaftFRFD [-rest]',model,lossfac,cf,sel)
def=fe_cyclicb('ShaftFRFMS [-rest]',model,def,damp,cf,sel)
fe_cyclic('ShaftPrep',model,def)
mdl=fe_cyclic('ShaftRing [-round N1 N2 N3 -autoclose N4]',rim1,rim2)
fe_cyclicb('ShaftLoad',model)
[def,mdl]=fe_cyclicb('ShaftTEig ORDERS',model)
```

OBSOLETE Basis [, -all, -norm, -int, -rb]

This command allows to build a set of modes :

- with the harmonics specified in an array $[\delta_1, \delta_2, ...]$,
- for the rotation vectors specified in a cell $\{[\omega_{1x}, \omega_{1y}, \omega_{1z}], [\omega_{2x}, \omega_{2y}, \omega_{2z}], ...\}$.

The general call is

```
[model,def]=fe_cyclicb('basis -all -int -norm -rb',sector,orders,omegas);
```

The output is a struct array containing the modeshapes. If only one output is required, the basis is added to the model as a `TR` field. The field `data` refers to the harmonic in column 2 and the rotation speed in column 3. The number of computed modes is controlled by the field `'info','EigOpt'` in the stack.

```
cf=demo_cyclic('testload disk 5 -nor reset') % reset file (rather than l
```

```
% Set of Cyclic Modes / Fixed Interface Modes
sector=stack_set(cf.Stack{'disk1'},'info','EigOpt',[5 4 0 11 1e-8]);
sector=stack_set(sector,'info','EigOptFixInt',[5 4 0 11 1e-8]);
sector=fe_cyclicb('basis -all -int -norm',sector,[0:3],...
{[0 0 0],[0 0 250]});
cf.model=sector;cf.def=sector.TR;
```

The `-all` option is used to get both modeshapes associated with a double eigenvalue in the case where harmonics are not 0 or half the number of sectors, when applicable. For more information on cyclic symmetry, please refer to [...].

The `-norm` option ensures that modes are orthonormalized in mass and stiffness because of convergence problems caused by the presence of double eigenvalues. This option is not required when the eigenvalue problem is solved with Nastran (`fe_eig` method 50). xxx discuss with EB

When the `-int` option is added, modes of the initial sector with its left and right interfaces fixed (clamped) are also computed and added at the beginnning of the output. These modes have a -1 in the field `data`. The computation parameters are specified in the `'info','EigOptFixInt'` stack entry. When this option but no harmonic are given, it computes only fixed interface modes.

The six rigid body modes of a cyclic symmetric structure are mono-harmonic with harmonic 0 (1T along and 1R around the axis of symmetry) and harmonic 1 (2T along and 2R around the other axes). Thus, the `-rb` option is used to compute two more flexible modes with harmonic 0 and four more flexible modes with harmonic 1.

OBSOLETE DiskEig [, -ord]

This command allows to compute the approximate modes of a reduced disk model built with the command `Reduce` of `fe_cyclicb`.

Command fails and is no longer maintained zzz see with arnaud

```
cf=demo_cyclic('testload disk 5 -nor') % reload model
fe_cyclicb('reduce 1 1 -int',cf);disk=cf.Stack{'diskmodel'};

% Mode computations
disk=stack_set(disk,'info','EigOpt',[5 30 1e3 11 1e-8]);
[def,disk]=fe_cyclicb('diskeig -ord',disk);
```

```
disp(def.data);
cf.def=fe_cyclicb('Display',cf,def);
```

The -ord option is used to identify the Fourier harmonic coefficient associated with each mode when dealing with the reduced model of a tuned disk. When dealing with a mistuned disk (whose modes are multi-harmonic), this returns the coefficient whose harmonic is the greatest.

DiskEngineLoad EO [, sel]

This commands builds a physical load, spatially mono-harmonic, on a specified set of nodes. If no selection is present, all nodes are used.

```
% Model Initialization
cf=demo_cyclic('testdisk 7 -blade -cf 2');

% External Load
Load=fe_cyclicb('DiskEngineLoad 3 r > 201',cf);
cf.def=Load;fecom showdefarrow;
```

OBSOLETE DiskFRF [, D, MS, -rest]

This command allows to build the Frequency Response Functions of a disk model, either full or reduced. A load and a set of observation DOF have to be defined and added to the model with fe_case. The frequency range is stored in the stack as a 'info','Freq' entry.

The general call is

```
xF=fe_cyclicb('DiskFRFD',disk,lossfac)
xF=fe_cyclicb('DiskFRFD -rest',disk,lossfac,cf,sel)
xF=fe_cyclicb('DiskFRFMS',disk,def,damp)
xF=fe_cyclicb('DiskFRFMS -rest',disk,def,damp,cf,sel)
```

The command FRFD assembles the matrices of the model then uses them to compute the response. An optional loss factor can be specified.

The command FRFMS synthetizes the response from a set of modeshapes. A damping ratio for all modes can be specified.

The option -rest recovers (xxxEB recovers, expands, interpolates, ...) the response computed on the reduced model to a given selection of physical DOF. Without selection, the response is expanded to the whole physical DOF set. This option must be disabled when dealing with a full disk model.

The following example builds both direct and sythetized responses of a reduced disk model to a 2EO excitation.

```
cf=demo_cyclic('testload disk 5 -nor') % reload model
% the call to fe_cyclicb basis is already done
fe_cyclicb('reduce 1 1 -int',cf);disk=cf.Stack{'diskmodel'};

% External Load
Load=fe_cyclicb('DiskEngineLoad 2',cf);
fe_cyclicb('DiskSeDefInit',cf);
Rload=fe_cyclicb('DiskSeLoad',cf,Load);
disk=fe_case(disk,'dofload','Blade_load',Rload);

freq=[1500:5:3000]';
disk=stack_set(disk,'info','Freq',freq);

% Restitution to Blade Dofs
tips=feutil('FindNode r>201',cf.mdl);
disk=fe_case(disk,'SensDof','Sensors',disk.DOF);

% Transfert Functions / Direct
xFd=fe_cyclicb('disk frfd -rest',disk,.002,cf,tips);

% Transfert Functions / Modal Synthesis
disk=stack_set(disk,'info','EigOpt',[5 50 -1e3 11 1e-8]);
[def,disk]=fe_cyclicb('diskeig',disk);
xFms=fe_cyclicb('disk frfms -rest',disk,def,.001,cf,tips);

% Response Plots
ci=iiplot;
XF=iicom('curveXF');
XF('Blade resp.D')=struct('w',xFd.data,'xf',xFd.def.','dof',xFd.DOF);
XF('Blade resp.MS')=struct('w',xFms.data,'xf',xFms.def.','dof',xFms.DOF);
iicom('subMagPha')
iicom(ci,'IIxOnly',{'Blade resp.D','Blade resp.MS'})
```

OBSOLETE DiskPlot

This commands provides a graphical representation of the generalied quantities (modeshapes or load) on a patch whose nodes represent the generalized DOFs.

The following example deals with such representation for both the modes and a 3EO excitation.

```
% Model Initialization
cf=demo_cyclic('testload disk 5 -nor reset')

% Model Reduction
fe_cyclicb('reduce 1 1 -int',cf);disk=cf.Stack{'diskmodel'};

% Mode Computations
disk=stack_set(disk,'info','EigOpt',[5 50 -1e3 11 1e-8]);
[def,disk]=fe_cyclicb('diskeig',disk);
fe_cyclicb('diskplot',3,cf,def);

% External Load
Load=fe_cyclicb('DiskEngineLoad 3',cf);
fe_cyclicb('DiskSeDefInit',cf);
Rload=fe_cyclicb('DiskSeLoad',cf,Load);
fe_cyclicb('diskplot',4,cf,Rload);
```

OBSOLETE Disk ... (internal commands)

DiskRest computes the Fourier Recovery Matrix associated with the specified harmonic. This function is used internally by fe_cyclicb.

DiskMatrices returns the reduced matrices assembled from the sector superelement matrices. When the option mk is specified, it returns the mass and stiffnes matrices only.

```
cf=demo_cyclic('testload disk 5 -nor reset') % reload model
fe_cyclicb('reduce 1 1 -int',cf);disk=cf.Stack{'diskmodel'};

Kr0=fe_cyclicb('diskmatrices',disk);
disk.il=[1001 1 1 250^2 250^4];
Kr=fe_cyclicb('diskmatrices mk',disk);
```

PolyAssemble [, noT] supports the computations of the coefficients of the matrix polynomial from the assembly of the stiffness matrices (including gyroscopic and nonlinear tangent stiffnesses) at the given values of the parameters. Three values are required.

The following example assembles the mass and full stiffness matrices at 0, 500 and 1000 rps, then computes the modes of the free sector at 250 rps.

```
% Model Initialization
cf=demo_cyclic('testdisk 5 -blade noK -nor -cf 2');
sector=fe_cyclicb('polyassemble noT',cf.Stack{'disk1'},[0 500 1000]);
[Case,sector.DOF]=fe_mknl('init',sector);
K={sector.K{1},feutilb('sumkcoef',{sector.K{2:4}},[1 250^2 250^4])};

% Mode Computations
def=fe_eig({K{1},K{2},Case.T,sector.DOF},[5 10 0 11]);
cf.model=sector;cf.def=def;
```

OBSOLETE DiskSeDefInit

This command initializes the restitution of the generalized modes computed with the DiskEig command on the physical DOF set. The restitution bases are stored in the Stack as a 'info','SeRestit' entry which contains the following fields:

- a field .DOF that contains the physical DOF set,
- a field .adof that contains the generalized DOF set,
- a cell .Restit where the first column gives the indices in the physical DOF vector associated with each sector, the second column gives the indices in the generalized DOF vector associated with each sector and the third column contains the transformation matrix from physical to generalized DOF coordinates.
- a cell .cGL that contains one matrix per sector which is a local to global frame transformation matrix.

The restitution is performed with the command SeDef of fesuper. In this command, a patch can be defined as a selection of elements so that the modeshapes are expanded only to the physical DOF of their nodes.

The following example builds a reduced model, computes a set of generalized modes, then the modeshapes are recovered on the blades only.

```
% Model Initialization
cf=demo_cyclic('testload disk 5 reset') % reset file (rather than load)

% Mode Computations
def=fe_eig(cf.mdl,[5 20 -1e3 11 1e-8]);

% Restitution
cf.def=fesuper('SeDef',cf,def);
```

OBSOLETE DiskSeLoad

This commands transforms an external load, expressed on the physical DOF set, into a generalized load, expressed on the generalized DOF set. It is necessary to initialize the restitution of the underlying disk model, since this transformation is the inverse operation.

```
% Model Initialization
cf=demo_cyclic('testload disk 5 -nor')
%xxx obsolete
%Model Reduction
fe_cyclicb('reduce 1 1 -int',cf);

% External Load
Load=fe_cyclicb('DiskEngineLoad 1',cf);
fe_cyclicb('DiskSeDefInit',cf);
Rload=fe_cyclicb('DiskSeLoad',cf,Load);
```

Reduce [, -int]

The Reduce command is used to generate a disk model from a set of cyclic modes associated with multiple harmonics. The general call is

```
[disk,SEsector]=fe_cyclicb('reduce NodeId0 EltId0 [-int]',model);
fe_cyclicb('reduce NodeId0 EltId0 [-int]',cf);
```

The second call uses directly a global model stored in the variable cf and stores the reduced model as an entry 'info','diskmodel' in cf.Stack.

If required, new starting points for numbering the generalized DOF and the associated elements are defined in the command string through the two integers NodeId0 and EltId0.

The initial set of modes has to be given as a field TR of the model. This operation can be performed directly by using the command Basis of fe_cyclicb, as shown in the following script.

```
% Model Initialization
cf=demo_cyclic('testload disk 5 -nor -cf reset');
fe_cyclicb('reduce 1 1 -int',cf);
disk=cf.Stack{'diskmodel'};
```

The reduction basis is built by separating right, left and interior motion from the cyclic modeshapes. When the `-int` option is invoked, fixed interface modes of the sector are added to the set of interior modes. For more information about this procedure, refer to [11].

nl_spring

Purpose *Non linear links/force modelization for time simulation*

Syntax

```
model=nl_spring('tab',model);
...= nl_spring('command', ...)
Opt=nl_spring('TimeOpt');
```

Description nl_spring supports non-linear connections and loads for transient analysis. Non linear springs between 2 DOF (see nlspring). loads which depend on DOF values (see DofKuva, DofV), springs between 2 nodes in different bases (see RotCenter), etc. ...). A full list of non-linearities is given below.

Standard non-linear simulations are handled by nl_solve. Below is a description of the inner mechanisms of a non-linear simulation with the non-linear toolbox.

After the non linearity definition, a proper TimeOpt is required to set the good fe_time calls to perform a non linear Newmark time integration. A default TimeOpt can be set using nl_spring TimeOpt.

The following steps are required for a time simulation

- Definition of non-linear properties. These are stored as pro entries of the model stack. The associated property function must handle non-linearities which is currently only the case for p_spring and p_contact.

  A non-linearity is always associated with elements or superelements (typically a celas element. A given group of elements can only be associated with a single non-linearity type.

  The information needed to describe the non linearity is stored in a .NLdata field.

- Model initialization using the an fe_case('assemble') call in fe_time, is followed by the building of a model.NL stack that describes all non-linearities of the model in a format that is suitable for efficient time domain integration. This translation is performed by the nl_spring NL command.

- Jacobian computation, see nl_spring NLJacobianUpdate.

- Residual computations are performed through mkl_utils. The nominal residual call is r=-fc; mkl_utils('residual', r,model,u,v,a,opt,Case);.

Supported non linearities

See nllist for supported non linearities, and nl_fun to add your own non-linearities.

ConnectionBuild

One can define a set of non linear links between 2 parts of a model using a call of the form
`[model,idof]=nl_spring('ConnectionBuild',model,data);`
`idof` is a second optionnal out argument. It returns the list of DOF concerned by links (it can be useful in order to reduce super elements keeping idof as interfaces DOF for instance). `data` contains all the information needed to define links. It is a 3 column stack like cell array. First column contains the string `'connection'`, the second the name of the non linear link described in the third column that contains a data structure with following fields:

`.Ci` define nodes to connect in first (`.C1`) and second component (`.C2`). It can be a vector of NodeId or a screw data structure (slave nodes of the model nodes via `RBE3` links, see see `sdtweb('fe_case#connectionscrew')`.

`.link` defines how to link component 1 to component 2. It is a 1x2 cell array. First cell defines the type of link (`'EqualDof'` or `'Celas'`) and the second gives information about the link. For celas link it is a standard element matrix row with 0 replacing NodeId :`[0 0 DofId1 DofId2 ProId EltId Kv Mv Cv Bv]`.

`.NLdata` (optional) defines non linearity associated to celas link. See the list in list of supported NL. If this field is not present or empty, only linear link is considered.

`.PID` (optional) is a 1x2 line vector that defines PID (second column of .Node matrix, see `sdtweb('node')` of connected node (1rst column for 1rst component).

`.DID` (optional) is the same as above, defining DID (third column of .Node matrix, see `sdtweb('node')` of connected nodes.

Following example defines a model with a cylinder and a hole in a block. The cylinder is linked to the block by 3 celas preserving the pivot link.

```
mo1=demosdt('demoConnection-vol'); % meshes models
mo1=fe_case(mo1,'fixdof','base','z==-1'); % clamps the cylinder base
r1=struct('Origin',[0.5 0.5 0.5],'axis',[0 0 1],...
          'radius',.1,'rtol',.01,'length',1,'Npt',-3,...
          'ProId',111,'planes',[]); % Cylinder-side
```

```
r1=nl_spring('ConnectionCyl',r1); % defines planes
r3=r1; r3.ProId=1; % Block-side
link={'connection','link1',struct('C1',r3,'C2',r1,...
        'link',{{'celas',[0 0 12345 12345 1000 0 1e9]}})}; % Defines c
[mdl,idof]=nl_spring('ConnectionBuild',mo1,link); % builds connection
cf=feplot(mdl); % displays in feplot
fecom promodelviewon;  fecom('curtab Cases','link1_2');
def=fe_eig(mdl,[5 20 1e3]); % computes the first 20 modes
if length(find(def.data<1e-3))>1; sdtw('_err','connection failed'); end
cf.def=def; fecom ColorDataAll % displays modes
```

See also `t_nlspring('2beam')` example.

ConnectionCyl

Define a cylinder connection.
`data=nl_spring('ConnectionCyl',data);`
`data` is a standard connection screw data structure (see `fe_caseg ConnectionScrew`) with following additional fields:

`.rtol` radius tolerance for cylinder selection.

`.length` length of the cylinder.

`.Npt` number of planes (equally distributed on the whole length). If Npt<0, ends of the cylinder are included in the connection points.

`.ProId` ProId of the elements containing nodes to connect.

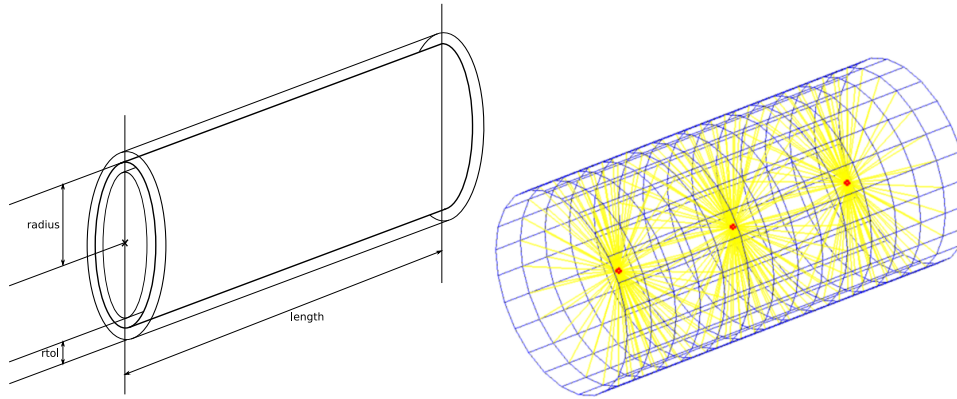

Figure 5.1: ConnectionCyl

InitV

```
q0=nl_spring('InitV',model,d0,RO);
```

InitV computes the initial static displacement and velocity associated to a DOF initial position and velocity. d0 is a data structure with field .DOF containing the DofId where initial value is applyed and .def containing initial displacement and velocity at this DOF. RO is a optional input argument data structure with following fields that define:

- .dt time step for time integration.
- .dq increment for initial vel computation.
- .Nv number of time steps to reach d0.def(1) (displacement is imposed as a $0.5(1-cos)$ time function on these time steps).
- .Np number of steps to stabilize at d0.def(1) and d0.def(1)+dq.

If input argument RO omitted, options are get from 'info' 'initvopt' Stack entry. If there is no such entry, InitV parameters are computed using -optim process (see below).

Displacement at q0 and q0+dq is obtained meaning the last Np/10 steps of each stabilization period, and initial velocity is computed from those 2 displacements to match d0.def(2) at d0.DOF.

`[q0,RO]=nl_spring('InitV-optim',model,d0);` can be used to find input parameters `RO`. Optimization of dt and Np is performed from given or default values. Parameters dq and Nv are kept at given or default value. First dt is optimized. dt is increased (multiplied by 4) until time integration of the `InitV` process diverge and last dt that leads to convergence is kept. Then Np is increased by 100 steps until the deformation is converged on the stabilization periods, that is to say that a criteria taking in account standard deviation/mean of the deformation and the ratio of the last Np/10 steps upon previous Np/10 steps on each Np period is less than a tolerance (2.0).

See 2beam example XXX.

NL

`model=nl_spring('NL',model)`
This command is used to build `.NL` field data for time integration from `NLdata` field in NL `p_spring` property entries in the input model Stack. The command option `-storefnl` can be used to specify the way of computing and storing a non linear effort associated to NL (for those which support it).

NLJacobianUpdate

`opt=nl_spring('JacobianCall')` returns the callback used to update or initialize the jacobian `ki` used in iterative methods. The said jacobian must take nonlinearities into account.

For the case of a non-linear spring, the most important gradient of the tabulated law `Fu` is added as stiffness between the 2 DOF to the stiffness matrix and the most important gradient of `Fv` to the damping matrix.

For non-linear iterations in a Newmark scheme, the Jacobian is given by

```
ki=(model.K{3}+kj)+ (opt(2)/opt(1))/dt*(model.K{2}+cj) + 1/opt(1)/dt^2*mo
```

Accepted command options, associated to variants of the call are

- `-TangentMdl` to return tangent model. It is assumed that `model.K(1:3)` correspond to M, C, and K (in this order). `u` and `v` variables of caller workspace can be needed.
- `-ener` to compute for each def stored in `model.d1` def structure (that is typically computed modes), some associated energies:

freq frequency in Hz.

damping damping ratio: $(\phi_j^T[C]\phi_j)/(2\omega_j)$.

enerK total strain energy: $\phi_j^T[K]\phi_j$.

enerC $\phi_j^T[K]\phi_j$.

NLlink-enerK strain energy for each NL link: $\phi_j^T[K_{NLlink}]\phi_j$.

NLlink-enerK for each NL link: $\phi_j^T[C_{NLlink}]\phi_j$.

SetPro

```
mdl=nl_spring('SetPro ProId i ParamName1 Value1 ...',mdl)
```

This command is used to change some nl_spring properties parameters. *i* is the ProId of corresponding p_spring property, *ParamName* the name of parameter to change (k for il(3), c for il(5) or the fieldname in NLdata) and *Value* the value to assign.

It is possible to define a new property by specifying an NLdata structure in third argument: `mdl=nl_spring('SetPro ProId i',mdl,NLdata)`. If the property already exists, it is overwritten by the NLdata structure provided.

Standard NLdata structures depend on the non-linear function, see nllist for more details. They can be obtained through the nl_function command db, see nl_fun for more details.

GetPro

```
pro=nl_spring('GetPro',model)
```

This command is used to get non linear properties in the model stack.

- Command option ID allows getting a specific non linear property by specifying its ProId.
- Command option type *''nl_fun''* allows getting the non linear properties of a specific type. See nllist for more details on types of non-linearities.

TimeOpt

This command returns usual default TimeOpt for non-linear simulations. By default the output is the same as the TimeOptNLNewmark presented below. See also fe_time for TimeOpt definition details.

Supported TimeOpt commands are

- TimeOptNLNewmark, or TimeOpt to obtain the TimeOpt for NLNewmark simulations. Use TimeOpt-gamma .51 to introduce numerical damping by directly giving gamma.
- TimeOptStat to perform static simulations (see also fe_time nl_solve).
- TimeOptTheta to perform time simulations with the θ-method (see fe_time). Numerical damping can be introduced using TimeOptTheta-alpha .05, the specified α value will be added to θ, so that the coefficient used in the simulations will be $\theta_1 = \theta + \alpha$.
- TimeOptExplicit to perform time simulations with the explicit Newmark scheme.

From standard fe_timesimulations, the following TimeOpt fields are added or modified

- Jacobian field is modified to take into account non linearities, see NLJacobianUpdat
- Residual field is modified to take into account non linearities, and to use mkl_utils to improve computation times, see sdtweb mkl_utils.
- AssembleCall field is modified, to perform non-linearities initialization after assembly. AssembleCall is the string passed to fe_case, generated by nl_spring('AssembleCall').
- OutputInit field is modified to also check non linearities and initialize non-linearities related outputs, this is a callback generated by nl_spring('OutputInit(
- FinalCleanUpFcn field is modified to perform cleanup on non linearities as well, this is realized through the ExitFcn command option of fe_simulfe_timeClea (see fe_timeTimeOpt), using '-ExitFcn"nl_spring(''fe_timeCleanUp'')"'.

TimeOutputOptions

Fine tuning of fe_time output can be achieved by specifying an 'info','OutputOptions' case entry.

Accepted fields for the OutputOptions structure are

- .FnlAllT if defined and equal to 1, non-linear loads are saved at all time steps.
- .FnlIiplot if defined and equal to 1, non linear loads are displayed in an iiplot figure as curve FNL. If the display timer associated with this figure does not stop automatically, you can stop it with cingui('TimerStop').

mkl_utils

Non linearities are treated by mkl_utils mex file. In order to make proper distribution of mkl_utils please send us result of:

```
ver
cd(fullfile(matlabroot,'bin',getenv('MATLAB_ARCH')))
!ls -lt *mkl*
```

For more details see sdtweb('mkl_utils').

rheo2NL

OBSOLETE. Use now nl_spring NL.
NL=nl_spring('rheo2NL',model,DOF,offset);
This command is used to convert rheological data into a structure of data understandable for NLforce command. DOF is the list of the DOF coherent with u and v arguments of NLforce command. Offset is optional. It is a structure of data with fields .DOF and .def that defined 0 reference for Fu and Fv tab laws.

tab

```
model=nl_spring('tab',model);
```

This command is used to convert formal rheological description data stored in model.Stack to a tabulated law description. The format is likely to change due to optimization of the compiled functionality in mkl_utils (see mkl_utils).

BlockSave,BlockLoad

Undocumented intermediate save of a time block for long simulations that do not fit in memory.

mkl_utils

Purpose

For detailed callback information see `sdtweb('nlspring_timeopt')` (section **??**)

`Residual` command is used to compute standard residue.
`mkl_utils('residual',r,model,u,v,a,opt,Case);` call modifies variable `r` in memory according to following standard residue computation.

```
r = model.K{1}*a + model.K{2}*v + model.K{3}*u - fnl -fc;
```

Typically in `fe_time` computations one has

```
opt.Residual='r=-full(fc);mkl_utils(''residual'',r,model,u,v,a,opt,Case);';
```

with `fc` the time load (resulting from `DofLoad` entries in model Case) and `fnl` is the sum of the non linear efforts (if any) computed directly by `mkl_utils` (`rotcenter`, `mocirc2`), in the non linear functions (see `sdtweb nl_fun`) or in `nl_spring`. `mkl_utils` then calls the adequate `nl_fun` function (`nl_spring` by default) automatically.

Model information specifically supported by the residual command are

- `opt.Rayleigh` if the field exists defines a global Rayleigh damping and `opt.Rayleigh(1)*m` is added to the residual.
- `model.K{2}` can be a data structure describing modal damping with following fields:

  `.def` : $M\Phi$ vectors as columns.

  `.data` : c_j modal damping coefficients as a vector. $c_j = 2\omega_j\zeta_j$.

  `.K` : optional additional damping matrix. This matrix must be in a mkl transposed `v_handle` format (use `v_handle('mklst',K)` to convert a matlab matrix to this format). Note that `model.K{2}.K` is taken in account for the jacobian computation whereas modal damping is not.

  Corresponding additional residue term is
  $\sum_j [M]\,\phi_j * c_j * \phi_j^T\,[M]^T * v$.

- `model.NL` can be a stack of non linearities. Column 3 provides a structure with the following standard fields

  `.type` a MATLAB function handle which is then called through `NL.type(NL,fc,model,u,v,` Also some older NL string giving the type name.

`.c` observation matrix for non linear displacements and velocities stored in row format obtained with `NL.c=v_handle('mklst',sparse(NL.c))`.

`.unl` pre-allocated memory for the result of `NL.c*u`. Must be consistent with the number of rows in `NL.c`. The computation is handled by `mkl_utils`.

`.vnl` if exists pre-allocated memory for the result of `NL.c*v`. Must be consistent with the number of rows in `NL.c`. The computation is handled by `mkl_utils`.

`.b` command matrix for non linear loads. At the end of the `NL.type` call it is expected that `NL.unl` contains the non linear component loads such that the residual becomes `r=r+NL.b*NL.unl`. The matrix is expected to be in transposed form `NL.b=v_handle('mklst',b');`. These conventions allow reuse of a `.c` matrix for command. Note that the sign conventions when using `unl` to return a non linear force are opposite to what is done when the result is added to `fc`, see `sdtweb nl_fun` to compare conventions.

–

Non linearities list

Purpose

List of supported non linearities. It is possible to create new ones (sdtweb nl_fun)

nl_inout

nl_inout is the more general non linearity, using observation and command matrix.

$f_{NL} = b \times f(C.u, C.v)$

The pro.NLdata structure has fields

.type	'nl_inout'
.lab	Label of the non linearity.
.b,.c	Command and observation definition. If the NLdata.DOF field is defined, placement of the observation matrix in done during the model assembly phase.
.Sens,.Load	Alternate form for .b, .c. xxx .Sens ell array of the form {SensType,SensData} where SensType is a string defining the sensor type and SensData a matrix with the sensor data (see sdtweb sensor).
.Load	data structure defining the command as a load (with .DOF and .def fields).
.Fu	defines the function of $C.u$ and $C.v$ defining the load. Multiple forms are possible. Currently a cell array of tables for Fu, a numeric curve ID for a curve in the stack, ... 2 predefined laws exist : Friction f *f* c0 *c0* and BumpStop dumax 0.02 k0 500 c0 0 dumin -0.02 k1 500 c1 0 (a bumpstop example can be found in sdtweb t_nlspring('BumpStop'). THIS IS NOT FULLY GENERAL. In particular c and b must have the same size for the moment. XXX need to be detailed. XXX.
.Elt	not available yet.
.unl0	Optional. Define an offset in observation (before applying Fu) as $unl = c * u + unl0$. It can be a vector giving direct offset (as many lines as c). It can also be a string defining what offset to apply : the only strategy implemented is unl0='q0' to remove observation of the static from observation at each time step. It can also be the name of a curve stored in model stack.

No jacobian is computed for this non-linearity. Maybe it will be needed in some cases.

nl_maxwell

nl_maxwell describes rheological models using stiffness and damping.

`.type`	`'nl_maxwell'`
`.lab`	Label of the non linearity.
`.Sens`	Observation definition. Cell array of the form `{SensType,SensData}` where SensType is a string defining the sensor type and SensData a matrix with the sensor data (see `sdtweb sensor`).
`.Load`	data structure defining the command as a load (with `.DOF` and `.def` fields).
`.SE`	superelement that defines the rheological model. Only matrices are used (`.K` field). Mass matrix is ignored. The `.DOF` field is unused and first DOF are assumed to be the observations defined, and following correspond to internal states.

Jacobian is computed using a Guyan condensation keeping only the observation (internal states are condensed) to obtain tangent damping and stiffness.
Internal states are integrated using an independent finite differences explicit scheme, with the same step of time as the main scheme, or a subsampling `opt.NLsteps` times. At the first residue computation, the initial internal states are computed according to initial condition in terms of displacements and velocities through a time integration until variation of speed between the 2 last computed steps is lower than `opt.RelTol`.
Force on the observation DOF (F), displacement (Qc) and velocity (dQc) of the internal DOF, displacement and velocity observations are stored in the NL output.

The command `nl_spring db` *`type`* is a database of generalized maxwell rheological models. *`type`* can be:

`zener` standard viscoelsatic model. Parameter *`k0`*, *`k1`* and *`c1`* can be given as a string of the form `db zener k0` *`k0`* `k1` *`k1`* `c1` *`c1`* in the command.

The example of the standard viscoelastic model is detailed here as an illustration. The standard viscoelastic model, also known as Zener model, is composed by a spring (K_0) in parallel with another spring (K_1) and a serial dashpot (C_1) as displayed figure 5.2.

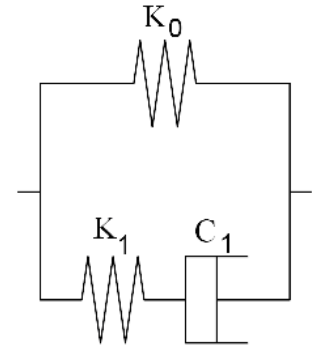

Figure 5.2: Standard viscoelastic model.

In the Laplace domain, the relation between the relative load and the relative displacement is given by

$$F(s) = K(s)X(s) = \frac{K_0K_1 + (K_0 + K_1))C_1s}{K_1 + C_1s} = K_0\frac{1 + s/z}{1 + s/p} \tag{5.3}$$

where p and z are respectively the pole and the zero of the model

$$p = \frac{K_1}{C_1} \tag{5.4}$$

$$z = \frac{K_0K_1}{(K_0 + K_1)\,C_1} \tag{5.5}$$

The maximum loss factor is

$$\eta_m = \frac{p - z}{2\sqrt{pz}} = \frac{1}{2}\frac{K_1}{\sqrt{K_0\,(K_0 + K_1)}} \tag{5.6}$$

and obtained for pulsation

$$\omega_m = \sqrt{pz} = \frac{K_1}{C_1}\sqrt{\frac{K_0}{K_0 + K_1}} \tag{5.7}$$

K_0 is the static stiffness of the model. Typically $K_1 = \frac{K_0}{2}$ and C_1 is defined so that the damping is maximal for the frequency of interest.

Following example considers $K_0 = 1000N/m$, $K_1 = 500N/m$ and $C_1 = 1.4Ns/m$. These parameters lead to a maximum loss factor of 20.14% for a frequency of 46.41Hz. The module and the loss factor are represented in figure 5.3.

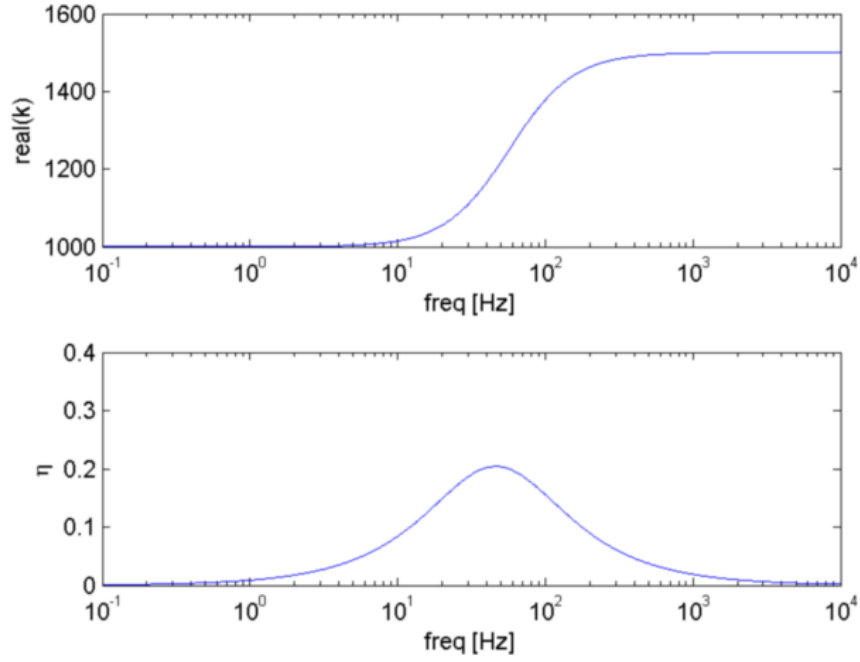

Figure 5.3: Module and loss factor.

Following example consists in a mass of 1e-2kg linked to the ground by the zener model. Initial displacement corresponding to a 1N load on the mass is imposed and then a time simulation is performed.

```
% parameters
 param.m=1e-2; param.dt=1e-4; param.N=1e3;
 param.k0=1e3; param.k1=param.k0/2; param.c1=1.4; % zener parameters
% define model
mdl=struct('Node',[1 0 0 0 0 0 0],...
           'Elt',[Inf abs('mass1') 0; 1 0 0 param.m  0 0 0]);
% define nl_maxwell data
data=nl_maxwell(sprintf('db zener k0 %.15g  k1 %.15g c1 %.15g',....
                        param.k0,param.k1,param.c1));
data.Sens{2}=1.03; % translation sensor defining nl_maxwell inputs
```

```
% define associated property
r1=p_spring('default'); r1=feutil('rmfield',r1,'name');
r1.NLdata=data; r1.il(3)=param.k0;
r1.il(1)=100; mdl=stack_set(mdl,'pro','zener',r1);
% define option for time integration
opt=nl_spring('TimeOpt');
opt.NeedUVA=[1 1 1];
opt.Follow=1; opt.RelTol=-1e-5;
opt.Opt(7)=-1; % factor type sparse
opt.Opt(4)=param.dt; opt.Opt(5)=param.N; % NSteps
opt.IterInit='model.FNL0=model.FNL+0;'; % needed for internal state xxx:
%opt.IterEnd='eval(opt.Residual)'; % to compute real FNL for current stat
% Initial state
r1=data.SE.K{3}\[1;0]; r1=r1(1); % initial displacement for 1N load
mdl=stack_set(mdl,'curve','q0',struct('def',r1,'DOF',1.03));
% Time computation
ci=iiplot(3);
def0=fe_time(opt,mdl); % compute

% The same but NL as a model
SE2=data.SE;
SE2.Elt(end+1:end+2,1:6)=[Inf abs('mass1'); 1 0 0 param.m 0 0];
SE2=fe_caseg('assemble -secdof -matdes 2 3 1 -reset',SE2);
r1=SE2.K{3}\[1;0]; %r1=r1(1);
SE2=stack_set(SE2,'curve','q0',struct('def',r1,'DOF',SE2.DOF));
def20=fe_time(opt,SE2); % compute
F20=SE2.K{2}*def20.v+SE2.K{3}*def20.def; F20=F20(1,:);
NL20=struct('X',{{def20.data {'LIN-F1';'LIN-Qc1';'LIN-dQc1';'LIN-unl1';'L
            'Y',[F20' (fe_c(def20.DOF,3.03)*def20.def)' (fe_c(def20.DOF,3
NL20.name='NLfromLIN';
iiplot(NL20)
iicom('IIxOnly',{'NL(1)','NLfromLIN'});
if norm(ci.Stack{'NL(1)'}.Y(2:end,:)-NL20.Y(2:end,:))/norm(NL20.Y(2:end,:
 sdtw('_err','something has changed')
end
```

DofKuva

`DofKuva` defines a non linear load of the form

$factor.Dof^{exponent}.K.V$

`.type`	`'DofKuva'`
`.lab`	Label of the non linearity.
`.Dof`	Dof of `Case.DOF`.
`.Dofuva`	`[1 0 0]` for displacement `Dof`, `[0 1 0]` for velocity and `[0 0 1]` for acceleration.
`.MatTyp`	Type of the matrix K (see `MatType`). Desired matrix is automatically assembled before time computation.
`.factor`	Scalar factor.
`.exponent`	Exponent of the DOF.
`.uva`	Type of vector V: `[1 0 0]` for displacement, `[0 1 0]` for velocity and `[0 0 1]` for acceleration.

For example one can take in account gyroscopic effect in a time computation with a NL of the form

```
model=stack_set(model,'pro','DofKuva1005', ... % gyroscopic effects
   struct('il',[1005 fe_mat('p_spring','SI',1) 0 0 0 0 0],...
   'type','p_spring','NLdata',struct(...
       'type','DofKuva','lab','gyroscopic effect', ...
       'Dof',1.06,'Dofuva',[0 1 0],'MatTyp',7,...
       'factor',-1,'exponent',1,'uva',[0 1 0])));
```

DofV

`DofV` defines a non linear effort of the following form (product of a fixed vector and a dof)

$Dof^{exponent}.V$

`NDdata` fields:

`.type`	`'DofV'`
`.lab`	Label of the non linearity.
`.Dof`	Dof of `Case.DOF`.
`.Dofuva`	`[1 0 0]` for displacement `Dof`, `[0 1 0]` for velocity and `[0 0 1]` f acceleration.
`.exponent`	Exponent of the DOF.
`.def`	data structure with fields `.def` which defines vector `V` and `.DOF` whi defines corresponding DOF.

nlspring

`nlspring` defines a non linear effort from rheological information (stop, tabulated damping or stiffness laws etc.) between 2 DOF.

To define a non linear spring, one has to add a classic `celas` element, linear spring

between only 2 DOF. The non linear aspect is described by associated properties as a 'pro' entry in the model Stack.

For the moment, the 2 DOF of the non linear spring have to be given in the .dof1 and .dof2 fields of this entry.

One can describe non linearity by a formal rheological description using one or more of following fields in the pro Stack entry:

- .But : [dumax k0 c0 dumin k1 c1]. For du from dumin to dumax, f=0. For du>dumax, k0 stiffness is applied to du-dumax, and for du<dumin, k1 stiffness is applied to du-dumin. Damping is not taken in account at this time (due to tabulated law strategy).
- .Fsec : [fsec,cpenal]. For dv<-fsec/cpenal or for dv>fsec/cpenal, f=fsec is applied. For -fsec/cpenal<dv<fsec/cpenal, f=cpenal*dv is applied. If omitted, cpenal=1e5.
- .K
- .C

This information will be converted in tabulated laws Fu and Fv using nl_spring tab (low level call that should be automatically called at the beginning of time computation).

One can also describe non linearity with a tabulated effort / relative displacement and effort / relative velocity law between the DOF (dof2-dof1), respectively in the Fu and Fv fields of the pro Stack entry. First column of Fu (resp. Fv) gives the relative displacements (resp. velocities) and second column gives the efforts. One can give a coefficient av factor of Fv depending on relative displacement as a third column of Fu. It can be useful to describe a non linearity depending on relative displacement and relative velocity. Force applied is F=av(du).Fv(dv). It is used in particular to describe damping in a stop (.But NL).

Following example performs a non linear time computation on a simple 2-node model:

```
model.Node=[1  0 0 0  0 0 0; 2  0 0 0  0 0 1];
model.Elt=[Inf abs('celas') 0 0;
            1 2 3 -3  0 1 0  10;   % linear celas
            1 2 3 -3  100 2 0 0;   % Associated to nl
];
```

```
model.Stack={'pro','celas1',...
 struct('il',[100 fe_mat('p_spring','SI',1) 1e3 1 0 0 0],...
        'type','p_spring',...
        'NLdata',struct('type','nlspring',...
                        'dof1',1.03,'dof2',2.03,...
                        'but',[1e-3 1e3 0 -4e-4 1e3 0]))};

model=fe_case(model,'FixDof','base',1);
model=fe_case(model,'DofLoad','in',struct('DOF',2.03,'def',1));
model=fe_curve(model,'set','input','TestStep 0.02');
model=fe_case(model,'setcurve','in','input');

opt=nl_spring('TimeOpt'); opt.Opt([4 5])=[1e-3 1e4];opt.NeedUVA=[1 1 0];
def=fe_time(opt,model);

% XXX need to update timeoptexplicit
if 1==2
opt=nl_spring('TimeOptExplicit'); opt.Opt([4 5])=[1e-3 1e3];opt.NeedUVA=[1 1 0]
d2=fe_time(opt,model);

figure(10);
subplot(211);plot(def.data,def.def');xlabel('Time [s]');title('displacement')
subplot(212);plot(def.data,def.v');xlabel('Time [s]');title('velocity')
subplot(211);hold on;
plot(d2.data,d2.def','r--');xlabel('Time [s]');title('displacement')
hold off;
subplot(212);hold on;
plot(d2.data,d2.v','r--');xlabel('Time [s]');title('velocity')
legend('Implicit','Explicit');
hold off
end
```

Following example deals with a clamped-free beam, with a bilateral bump stop at the free end. XXX: image

```
% define model:
L=1; b=1e-2; h=2e-2; e=1e-3; % dimensions
model=[];
model.Node=[1  0 0 0  0 0 0; 2  0 0 0  L 0 0];
model.Elt=[Inf abs('celas') 0 0;
           2 0  2 0     100  1 110 0;   % linear celas
```

```
            Inf abs('beam1') 0 0;
            1 2  1 1    0 1 0 0
            ];
model=feutil(sprintf('RefineBeam %.15g',L/20),model);
model=fe_case(model,'FixDof','base',1); % clamps 1st end
model=fe_case(model,'FixDof','2D',[0.03;0.04;0.05]); % 2D motion
% model properties:
model.pl=m_elastic('dbval 1 steel');
model.il=p_beam(sprintf('dbval 1 BOX %.15g %.15g %.15g %.15g',b,h,e,e));
% Bump stop NL:
model=stack_set(model,'pro','celas1',...
        struct('il',[100 fe_mat('p_spring','SI',1) 1e-9 0 0 0 0],...
               'type','p_spring',...
               'NLdata',struct('type','nlspring',...
                               'dof1',2.02,'dof2',0,...
                               'but',[0.02 5e2 0 -0.02 5e2 0],... % gap k
                               'umin',3)));
if 1==1
 model=fe_case(model,'DofLoad','in',struct('DOF',2.02,'def',50));
 model=fe_curve(model,'set','input','TestStep 0.02');
else
 f=linspace(12,18,3);
 model=fe_case(model,'DofLoad','in',struct('DOF',2.02,'def',1));
 model=fe_curve(model,'set','input',sprintf('Testeval cos(%.15g*t)',f(1)*
end
model=fe_case(model,'setcurve','in','input');
% Time computation:
opt=nl_spring('TimeOpt'); opt.Opt([4 5])=[1e-3 1e4];opt.NeedUVA=[1 1 0];
def=fe_time(opt,model);
```

RotCenter

The `Rotcenter` joint is used to introduce a penalized translation link between two nodes A and B (rotation DOFs of NL entry are ignored), where the motion of A is defined in a rotating frame associated with angle θ_A and large angle rotation $R_{LG}(\theta_A)$. The indices G and L are used to indicate vectors in global and local coordinates respectively.

The positions of nodes are given by

$$\begin{array}{rcl} \{x_A\}_G & = & [R_{GL}]\left(\{p_A\}+\{u_A\}_L\right) \\ \{x_B\}_G & = & \left(\{p_B\}+\{u_B\}_G\right) \end{array} \tag{5.8}$$

which leads to expressions of the loads as

$$\begin{array}{rcl} \{F_A\}_L & = & [R_{LG}]\left(K\left(\{x_B\}_G-\{x_A\}_G\right)\right) \\ \{F_B\}_G & = & K\left(\{x_A\}_G-\{x_B\}_G\right) \end{array} \tag{5.9}$$

To account for viscous damping loads in the joints, one must also compute velocities. Using (2.2), one obtains

$$\begin{array}{rcl} \{\dot{x}_A\}_G & = & [R_{GL}]\left(\{\dot{u}_A\}_L+\{\omega(t)\}\wedge\{p_A+u_A\}_L\right) \\ \{\dot{x}_B\}_G & = & \{\dot{u}_B\}_G \end{array} \tag{5.10}$$

Velocity computations are currently incorrect with u_A ignored in the rotation effect. So that viscous damping loads can be added

$$\begin{array}{rcl} \{FC_A\}_L & = & [R_{LG}]\left(K\left(\{\dot{x}_B\}_G-\{\dot{x}_A\}_G\right)\right) \\ \{FC_B\}_G & = & K\left(\{\dot{x}_A\}_G-\{\dot{x}_B\}_G\right) \end{array} \tag{5.11}$$

For a linearization around a given state (needed for frequency domain computations or building a sensor observation matrix),

$$\left\{\begin{array}{c} q_{AG} \\ q_{BG} \end{array}\right\} = \left[\begin{array}{cc} R_{GL} & 0 \\ 0 & I \end{array}\right] \left\{\begin{array}{c} q_{AL} \\ q_{BG} \end{array}\right\} \tag{5.12}$$

In global basis, stiffness matrix of a `celas` link is given by

$$k\left[\begin{array}{cc} I & -I \\ -I & I \end{array}\right] \tag{5.13}$$

which leads to the following stiffness matrix

$$\left[\begin{array}{cc} R_{GL}^T & 0 \\ 0 & I \end{array}\right] k \left[\begin{array}{cc} I & -I \\ -I & I \end{array}\right] \left[\begin{array}{cc} R_{GL} & 0 \\ 0 & I \end{array}\right] = k \left[\begin{array}{cc} I & -R_{GL}^T \\ -R_{GL} & I \end{array}\right] \tag{5.14}$$

where q_A DOFs are in the local basis (motion relative to the shaft in its initial position) and q_B are in the global frame.

`data` describing this link is stored in model stack as a `p_spring` pro entry. Stiffness and damping are stored respectively as 3rd and 5th column of the `data.il` field (standard linear spring, see `sdtweb('p_spring')`).

`NDdata` fields:

`.type` string `'RotCenter'`.

`.sel` a `FindElt` command to find `celas` of `RotCenter` type.

`.k` this field should not be used. `.JCoef` field should be used instead and has priority. Stiffness used for Jacobian computation. Damping is not taken in account in Jacobian in this case.

`.JCoef` coefficient of celas stiffness and damping for jacobian computation. Default is 1.

`.drot` the rotation DOF.

`.lab` label.

nl_rotCenter

This non linearity can be used to connect 2 points A and B, where the motion of A is defined in a rotating frame associated with angle θ_A and large angle rotation $R_{LG}(\theta_A)$. More generally A and B are no real nodes but defined implicitely as observation matrices. `nl_rotcenter` is an extension of `RotCenter` documented above, using observation matrices which is more general.

`.type` string `'nl_rotcenter'`.

`.sel` a `FindElt` command to find elemnts asosciated to the NL link (XXX really used ?).

`.JCoef` coefficient of celas stiffness and damping for jacobian computation. Default is 1.

`.drot` the rotation DOF.

`.lab` label.

`.Weights` (optional) Weight of the stiffness in a pivot link (in fact computed force is multiplied by the weight factors before being applied so that the sum of weight coef divided by number of points by pivot should be equal to 1).

`.Stack` Stack of cta coupling. Of the form `{'cta', 'name',{r1,r2}}`, where `'cta'` is a constant string defining the type of the link, `'name'` a string containing the name of corresponding links. `r1` is the observation in the first (rotating) part. It is a data structure with fields `.Node` defining the nodes involved,

.cta defining the observation matrix, .DOF defining corresponding DOF (as many columns as in .cta) and .SeName defining as a string the name of the superelement where cta is defined (XXX if omitted, it is assumed that DOF and cta are defined on the model.DOF - no superelement -). r2 is the same for the non rotating part.

An example can be found in t_nlspring 2beam (XXX this must be generalized to make it separated from psa08).

Temporary strategy was to define prior a RotCenter link, then convert it to the nl_rotcenter link using mdl=psa08('nl_rotcenterFix',cf.mdl,R0); command. This should no longer be used XXX.

Default uses the damping and stiffness defined in the il field of the p_spring pro entry to model a linear spring/damper between the 2 parts (stiffness il(3) and damping il(5)).

Defining a xb parameter, the Excite NONL law will be applied instead of the spring/damper. Parameter that are to be defined are

.xb Radial clearance.

.kb Stiffness at radial clearance.

.cb Damping at radial clearance.

Stiffness and damping at initial position are given in corresponding p_spring properties il(3) and il(5). For example:
cf.mdl=nl_spring('setpro ProId 103 k 371 c 2000e-3 xb 0.03 kb 37100 cb 5',cf.mdl);

One can also use a squeeze-film type law defining a .muRL3 field, with following parameters (XXX this need to be reactivated and updated) :

.muRL3 oil viscosity*bearing radius*bearing length^3.

.boundary Boundary condition (0=Sommerfeld,1=Gumbel(p¿0)).

.N_theta Number of integration point.

Defining a .RelF field, one can define a sub call to another non linearity to compute the force resulting from the excentricity computed by nl_rotcenter. In

that case, the .RelF field contains the NLdata field of the sub NL to be called. Following example (XXX need to geenralize psa08 ShaftRebuild and ShaftTime XXX) define a zener link in the bearing of simple 2beam example:

```
cf=feplot(2); t_nlspring('2beam-build')
psa08('ShaftRebuild',cf) % define large rotation DOF
psa08('ShaftTime 2000 1 1e-5 0',cf) % Omega(rpm), N_rot, dt, theta0
mo1.il(2,6)=0; % No gyroscopic because of shaft beam1
opt.RotFollow=1;% Follow velocity and not iterations
r1=stack_get(mo1,'pro','celas103','GetData');
% define zener
NLdata=nl_maxwell('db zener k0 125e3 k1 50e3 c1 0.4');
r1.NLdata.RelF.type='nl_maxwell'; r1.NLdata.RelF.NLdata=NLdata;
mo1=stack_set(mo1,'pro','celas103',r1);
def=fe_time(opt,mo1);
```

rod1

XXXEB : following needs to be checked The rod1 non-linear connection is a simple penalized rigid link. One considers two nodes A and B (see figure 5.4).

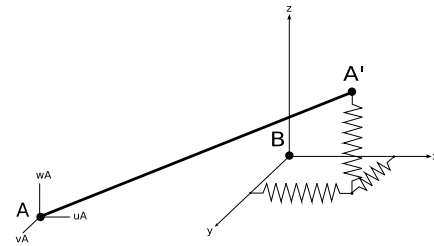

Figure 5.4: Large rotation rod functional representation.

Currently, one can introduce masses at points A and B. mass2 elements should be used to account for the actual position of the center of gravity.

The global non linear load associated with the rod is thus

$$F_{rod} \quad = \quad k_r \left(\| \{x_B - x_A\} \| - L_0 \right) \frac{\{x_B - x_A\}}{\| \{x_B - x_A\} \|} \tag{5.15}$$

which accounts for a load proportional to the length fluctuation around L_0 (penalized rod model).

When linearizing, one considers a strain energy given by $k_r \|q_B - q_{A'}\|^2$ with the motion at node A' being related to the 6 DOFs at node A by

$$\{q_{A'}\} = \begin{bmatrix} I & \left[\vec{AB}\wedge\right] \\ 0 & I \end{bmatrix} \{q_A\} \tag{5.16}$$

Node A node is free to rotate. The linearized stiffness thus corresponds to an axial stiffness in the direction of the rod. The computation of the stiffness is however based on the current position of the extremity nodes, a difficulty in model manipulations is thus to translate these nodes.

`data` describing this link is stored in model stack as a `p_spring` pro entry. Stiffness and damping are stored respectively as 3rd and 5th column of the `data.il` field (standard linear spring, see `sdtweb('p_spring')`). NL information is stored in the `data.NLdata` field which has itself following fields :

- type : string `'rod1'`.
- sel : a `FindElt` command to find associated `celas` of `rod1` type ((`'proid100'`).
- ulim : build tabulated law from -ulim to ulim. Default is 1e3.
- lab : label.

`nl_modaldmp`

Implementation of modal damping. Although modal damping is not a non-linear feature in itself, its implementation needs makes it behave as a non-linearity.

The concept is to provide shapes defined on a part of a model with associated damping ratios. `nl_modaldmp` handles the kinematic projection on the model which can contain superlements. In the case where superelements are used and concerned with modal damping, the shapes provided must be written on the physical DOF of the superelements.

The set of shapes must be stacked in model with a valid `ID` field. It is a common deformation SDT data structre (see `sdtweb def`), with an additional `.ID` field. The `.data` field is equivalent to the ones of complex modes (see `fe_ceig`). It is a matrix of two columns respectively giving the frequency and the target damping ratio for each mode.

Since modal damping implies a modal sensor, the features performs both by default. It is however possible to simplify it as a pure modal sensor. The theory around modal sensing/damping can be found in [12].

The pro.NLdata structure has fields

- type: string 'nl_modaldmp'.
- CurveId: the curve ID stacked in model which provides the shapes and their damping ratios.
- SensorOnly: to use the feature only as a modal sensor in a def data structure.

The NLdata structure generation can be integrated using an nl_modaldmp('db') call. See sdtweb nl_spring#setpro for this integration. This is used in transient simulations, and in complex mode computations, see nl_solve.

nl_contact

See p_contact.

Creating a new non linearity: nl_fun.m

Purpose

The structure of nl_spring allows creating any new non-linearity through the use of a dedicated function, named nl_fun.m. This function which non-linearity name will be fun, will be automatically called by nl_spring for classical operations.

The function structure has been designed to comply with specific needs. Standard calls have been defined, which are detailed below:

- **Residue computation**, called by mkl_utils (sdtweb mkl_utils), must ouput the entry force minus the non linear force computed. The call performed is

  ```
  nl_fun(r2,fc,model,u,v,a,opt,Case)
  ```

  This call is low level and must modify fc using sp_util('setinput') as fc-fnl where fnl is the non linear force computed. Note that this is the only possible call for nargin==8. Note that mkl_utils allows a formalism with precomputed observations, using fields unl.

- **Jacobian computation**, must output the tangent stiffness and tangent damping matrices associated to the non linearity. The call performed is

  ```
  [kj2,cj2]=nl_fun(NL,[],model,u,v,[],opt,Case,RunOpt);
  ```

  This call must output either empty matrices if no tangent nor Jacobian matrix is associated to the non linearity, or matrices expressed on the DOF vector of Case.DOF. The first matrix is the tangent stiffness matrix, the second on is the tangent damping matrix.

- **Initializations** for fe_time, must initialize the model non-linearity for non linear forces computation

  The call must generate the non linearity stored in model.NL, it can optionally generate non linear DOF and labels. The call performed is of the type.

  ```
  NL=nl_fun('init',data,mo1);
  ```

  NL is a struct containing at least the field type with the nl_fun handle (*e.g.* NL.type=@nl_fun). data contains the Stack,pro entry, and mo1 is the model, named mo1 where the call is performed.

- **ParamEdit** returns the ParamEdit string allowing integrated parameters interpretation (for internal SDT use).

  The call performed is of the type.

```
st=nl_fun('ParamEdit');
```

- **db** returns default NLdata fields for a non linearity. This allows integrated building of non-linearities in a model. This function can call ParamEdit to allow interactive setup.

  This call must return a NLdata field and is of the type

  ```
  NLdata=nl_fun('db data 0');
  ```

- **Renumbering** capability, must return the non-linearity written for the new renumbered nodes, elements, dof, ...

  The call performed (by feutilbfor example) is of the type

  ```
  NL=nl_fun('renumber',NL,nind);
  ```

  nind is the renumbering vector.

The designed nl_fun template is given in the non-linear toolbox, sdtweb nl_fun.m#1. It is a functional non linear function, computing a zero non linear force. The definition of a non linearity using nl_fun in a standard SDT model is given in the following.

```
% A standard SDT model
model=struct('Node',[1  0 0 0  0 0 0; 2  0 0 0  0 0 1],...
             'Elt',[Inf abs('celas') 0 0;
             1 2 3 -3  0 1 0  10;   % linear celas
            ]);
% Define a non linearity of type nl_fun
model=nl_spring('SetPro ProId 100',model,nl_fun('db data0'));
%Equivalent to
% model=stack_set(model,'pro','nl_fun',...
% struct('il',[100 fe_mat('p_spring','SI',1)],...
%        'type','p_spring',...
%        'NLdata',struct('type','nl_fun','data',[])));

% Define the case
model=fe_case(model,'FixDof','base',1);
model=fe_case(model,'DofLoad','in',struct('DOF',2.03,'def',1));
model=fe_curve(model,'set','input','TestStep 0.02');
model=fe_case(model,'setcurve','in','input');
% Define the TimeOpt and compute the solution
opt=nl_spring('TimeOpt'); opt.Opt([4 5])=[1e-3 1e4];opt.NeedUVA=[1 1 0];
def=fe_time(opt,model);
```

nl_solve

Purpose *Integrated non linear simulations*

Description The simulation of non linearities require special handling in SDT, which is packaged in the non linear toolbox. This function aims at performing classical studies, such as done by fe_simulfor classical SDT models with this special handling.

See nllist for the list of supported non linearities.

Static

To compute the static state of a model with non-linearities.

```
q0=nl_solve('static',model);
```

It is possible to use custom fe_time simulation properties using the model stack entry info,TimeOptStat. See nl_spring TimeOpt for fields and defaults.

It is possible to use as command option any field from the usual static simulation option, see sdtweb nl_spring#TimeOpt to have more details. *E.g.* To redefine on the fly the maximum number of iteration, one can enter [q0,opt]=nl_solve('static maxiter 100',model);.

By default, the StaticNewton algorithm implemented in fe_time is called.

An Uzawa algorithm is also implemented in nl_solve, under the method static nl_solve uzawa. This algorithm is very different from the StaticNewton one since here the solution is not incremented but fully re-computed at each iiteration. This is useful when some non-linear forces do not derive from potentials. Command StaticUzawa can be used in nl_solve – to access it: q0=nl_solve('static Uzawa',model);.

Mode

The definition of modes for non-linear models is not straight forward. This command aims at computing tangent modes as function of a non-linear model current state. The resolution thus concerns a linear model with tangent stiffness, damping matrices corresponding to the model current diplacement, velocity, accleration state. The eigenvalue solvers used are then fe_eigfor real modes and fe_ceigfor complex modes.

By default, modes tangent to a static state are computed. A static simulation is performed to produce a model state from which tangent matrices are computed. It is also possible to compute tangent modes at specific instants during a transient simulation, at SaveTimes instant, and to store frequency/damping data and deformations. Command options a A set of command options allows precising the mode computation wanted and the output.

Accepted command options to control the model computation itself are

- -cpx for complex mode computation (default is real mode computation).
- -stat for mode computation based on a static state (typically after an fe_timestat simulation).
- -time for mode computations during a transient simulation (exclusive with the default -stat option).
- -traj for mode computations based on states provided as an additional argument. xxx
- -skip skips fe_timesimulations and use the data provided in model to compute the modes. For a static computation, the model stack entry curve, q0 will be used. If empty, a zero displacement static state will be used. For a transient mode computation, the model stack entry curve,TSIM will be used, if empty an error will occur.

For options -skip and/or -traj, a deformation is/can be provided (either the time deformation, or a static deformation in the model stack). This deformation is stacked in the working model under the name curve,DEF_TRAJ and can be accessed during all initialization phases.

Accepted command options to control the output format are

- -dataOnly to save only the frequency, damping data (does not store the deformation field). The output is then under a frequency tracking curve in the iiplotformat.
- -fullDOF to output the deformation fields restituted on the unconstrained DOF.
- -alpha (requires -cpx) to also output the real mode participation to the complex modes. This is in fact the projection of the complex modes on the real mode basis.

- `-real "ModeBas"` (requires `-cpx`) to specify a particular real mode basis on which the complex modes will be computed. The real mode basis is supposed to be stored in the model stack entry `curve, ModeBas`.

Sample calls to extract tangent modes are given below.

```
def0=nl_solve('Mode',model)
def0=nl_solve('Mode-stat-fullDOF',model);
defT=nl_solve('Mode-time',model);
hist=nl_solve('Mode-time-dataOnly',model);
histC=nl_solve('Mode-cpx-time-dataOnly',model);
defC=nl_solve('Mode-cpx-time-alpha-real''MyBas''-fullDOF',model);
def1=nl_solve('Mode-skip-fullDOF',model);
```

SineSweep

xxx

nl_mesh

Purpose *Integrated mesh modifications for non-linear applications*

Description Some non-linearities require surface/volume remeshing (*e.g.* definition of conforming interfaces for contact) or adpations (*generation of thin interface layers*). This function regroups such functionalities. Mesh generation are performed by fe_gmsh(interface to gmsh) and fe_tetgen (interface to tetgen, see help fe_tetgen).

Conform

The Conform call is an integrated call to generate comforming meshes between two facing interfaces. The command generates a conforming surface mesh of the face to replace, merges it with the conform mesh of the second interface, replaces the model face mesh and remeshes the model volume to yield a new equivalent volume with a conform face mesh.

```
mo1=nl_mesh('conform eltsel"FindElt"',model,sel);
% sel={eltSelToReplace eltSelForReplacement;...}
```

model is a standard SDT model. sel is a cell array containing in each line two FindElt commands specifying the element selection face to remesh and the element selection face to use for the conforming interface for replacement.

- Command option eltsel allows specifying in a string a FindElt command restraining the working area in the original model.
- Command option smartSize allows generating a conforming mesh with a coherent mesh characteristic length.
- Command option gmsh allows using gmsh to mesh the final volume.
- Command option tetgen allows using tetgen to mesh the final volume (by default).
- Command option output asks to output the generated mesh in a .mat file.
- Command option OrigContour asks to keep original positions of mid-nodes of the quadratic faces delimiting the volume to remesh. This may however yield mesh wraping problems when the face to remesh is much coarser than the mesh trace to place for conformity.

- Command option mergeTo allows specifying a FindElt selection command in a string to replace the mesh on another model selection than the one used to generate the conforming interface (which uses eltsel.
- It is also possible to provide additional arguments, which will be passed the the nl_meshcover call performed in the procedure.

Limitations: The Conform call only supports generation of conforming interfaces when one interface contour fully contains the other interface contour. Handling of more complex contour configurations has not been implemented. Besides, this function has been designed to handle planar surfaces. Additional operation to work on non planar surfaces are left to the user (*e.g.* pre/post projections of the surfaces on a plane).

Cover

The Cover call is degined to mesh the interstice between two closed planar contours, when one fully contains the other. The call is performed as

```
[newModel,opt,largeContour]=nl_mesh('cover',model,{eltsel_large,eltsel_small});
```

model is a standard SDT model. Variables eltsel_large and eltsel_small are FindElt calls defining the element selection of the respectively large surface and small surface (the small being contained in the large).

The output newModel is the mesh generated from the surface contours.

opt outputs additional information about the mesh generation, it is a struct containing fields .NodeAdd specifiying the potential nodes added in the interstice space meshed,.nodeEdgeSel1 specifying the NodeId of the nodes located on the eltsel_large contour, .nodeEdgeSel2 specifying the NodeId of the nodes located on the eltsel_small contour, and .tname the name of the temporary file containing the generated mesh.

largeContour provides the original contour in beam elements of the eltsel_large selection.

- Command option merge allows merging the interstice mesh with the inner mesh of the eltsel_small selection.
- Command option quad allows generating proper quadratic meshes.
- Command option smartSize allows generating an interstice mesh with a characteristic length in coherence with the contour mesh lengh.

- Command option lc*val* allows setting the characteristic length to *Val* to the interstice mesher.
- Command option algo ''*name*'' allows specifying the meshing algorithm *name* to the gmsh mesher. See the gmsh documentation for more information.

Replace

The call Replace is designed to replace parts of a model face mesh with new given face meshes. It is performed as

```
model=nl_mesh('replace',model,nodesToReplace,NewModel,nodeIDtoKeep)
```

model is a standard SDT face (shells) model. nodesToReplace is a cell array containing vectors of NodeId specifying the areas to be replaced. NewModel is a cell array containing the new face models which will be merged to the mesh skin in coherence with the removed elements (specified by nodesToReplace). nodeIDtoKeep is an optionnal argument specifying NodeId of the original model for nodes whose NodeId must not change in the transformation.

- Command option setMat allows defining a specific MatId to the output mesh.
- Command option setPro allows defining a specific ProId to the output mesh.
- Command option keepNoCheck in combination with the use of a third argument nodeIDtoKeep assumes the nodes numbering is correct and forces the nodes original numerotaion without check.

Contour

Call ContourFrom generates SDT beam1or beam3 contour models for CAD definitions. All formats readble by gmsh can theoretically be used. Only the .geo, .stp and .igs are tested.

Since .geo files can contain geometric yet undiscretized objects, a 1D meshing pass is performed with gmsh to provide an SDT contour model. This is not supported for other file types.

```
model=nl_mesh('contourFrom','file.stp'); % not specifying the type
```

Call Contour generates an SDT face mesh from an SDT beam11 or beam13 contour.

```
model=nl_mesh('contour',model);
```

`model` is an SDT beam model defining a closed contour.

- Command option `lc`*`val`* allows specifying a characteristic length for gmsh.
- Command option `lcmin`*`val`* allows specifying a minimal characteristic length for gmsh.
- Command option `quad` allows generating quadratic meshes.
- Command option `keepNode` asks to keep the original contour `NodeId` for the `contour` command.
- Command option `diag` asks to output the gmsh log file for diagnostic problems.
- Command option `single` tells `nl_mesh` that a single coutour is defined. This is usefull when several closed contours are defined since it is impossible to automatically decide whether each contour is independant or if they define a single complex contour.
- Command option `group`*`val`* is used in combination to the `s`ingle command option. This allows specifying which contour group will be meshed, while other possible contours will define holes.
- Command option `algo''`*`val`*`''` allows specifying which algorithm `gmsh` must use (this depends on the `gmsh` version, report to the `gmsh` documentation for more details).
- Command option `AllowContourMod` allows `gmsh` adding nodes on the contour provided. By default `gmsh` is forced not to add nodes to the lines defining the contour to mesh.

Rivet

This command generates rivet drills in a specified contour. A model containing a beam contour can be provided, or an `EltSel` string generating a surface selection (see section and the `selface` option) on a bigger model. A data structure providing the origins, and rivet radiuses and washer (or rivet head radiuses). The mesh generated between both radiuses is structured.

The data structure must contain fields

- `Orig` providing the rivet centers in an `[x y z;...]` matrix.
- `radHole` providing the rivet hole radius, either a scalar if all rivets have the same radius, or a line vector providing each rivet radius separately.
- `radWash` providing the rivet washer (or head) radius, either a scalar if all rivets have the same washer radius, or a line vector providing each rivet washer radius separately.

and can optionnaly contain fields

- `plane` To directly provide the contour plane normal to define the drillings, in an `[nx ny nz; ...]` matrix.
- `Ns` To define the number of mesh segments in the rivet to washer radius area (default 10), either a scalar if all rivet heads have the same properties, or a line vector defining the property for each rivet separately.
- `Nr` To define the number of mesh radial nodes in the rivet to washer radius area (default 2), either a scalar if all rivet heads have the same properties, or a line vector defining the property for each rivet separately.

- Command option `MatId`*val* allows setting the modified mesh `MatId` to *val*.
- Command option `ProId`*val* allows setting the modified mesh `ProId` to *val*.
- Command option `-fill` outputs in second argument a compatible mesh of the rivet bores.

Following example meshes a rectangular contour with a few rivet drillings inside.

```
% Generate a global contour
model=struct('Node',[...
 1 0 0 0  0  0 0;
 2 0 0 0 10  0 0;
 3 0 0 0 10  2 0
 4 0 0 0  0  2 0], 'Elt',[]);
model.Elt=feutil('ObjectBeamLine 1 2 0 2 3 0 3 4 0 4 1',model);
model=feutil('refinebeam .2',model);
%feplot(model)
```

```
% define rivet positions, eventually planes
RO=struct('Orig',[ 3 1 0;6 1 0;9 1 0],...
 'radHole',[.2;.2;.2],...
 'radWash',[.8;.8;.8]);

model=nl_mesh('Rivet',model,RO);
cf=feplot(model);
```

GmshVol

This call integrates the generation of a volumic mesh from a face mesh with gmsh.

```
model=nl_mesh('GMSHvol',model);
```

`model` is a standard SDT face mesh model.

- Command option `setmat` allows specifying a specific `MatId` to the output mesh.
- Command option `setpro` allows specifying a specific `ProId` to the output mesh.
- Command option `keepFaces` asks to keep orignal `NodeId` of the nodes located on the face mesh.
- Command option `lc` specifies a characteristic length for gmsh.
- Command option `clmin` specifies a minimal mesh length for gmsh.
- Command option `clmax` specifies a maximal mesh length for gmsh.

ExtrudeLayer

This command generates a non trivial extrusion of a face mesh follwing the face normal at each node, to generate a volumic layer.

```
model=nl_mesh('ExtrudeLayer thick Val',model);
```

`model` is an SDT model with shell elements (a surface definition).

Command option `thick` specifies the extrusion thickness. Command option `setmat` allows specifying a specific `MatId` to the output. Command option `setpro` allows specifying a specific `ProId` to the output.

StackClean

This call cleans up a model stack when mesh modifications have been performed. It cleans up stack entries definition that became incoherent with some mesh modifications.

```
model=nl_mesh('StackClean',model);
```

Command option rmuns removes stack entries that could not be sorted out. Command option rmmod removse stack entries affected by the model modifications.

See also celas, p_spring, fe_gmsh

spfmex_utils

Purpose

`OfactOptim`

This command can be used to set spfmex parameters in order to optimize computation speed for factorization and / or solving.

```
spfmex_utils('OfactOptim',ki,RO,ofact(1,'lu'));
```

`ki` is the matrix that is used for the optimization. `RO` is a data structure defining options with following fields:

`.nCompt` Number of computation for result averaging.

`.maxDomain` Max size of blocks of the elimination tree (fraction of matrix size).

`.maxZeros` Max number of zeros in the blocks of the resolution tree (fraction of matrix size).

`.refineStep` Number of step to refine the optimal parameter pair found in the first step. Command option `-refine` must be added to perform the refine step.

The last argument `ofact(1,'lu')` is needed in order to call directly `spfmex_utils`.

Available command options are

`-setopt` use default `RO`.

`-refine` performs refine step for optimal search.

`fact` to benchmark factorization step.

`solve` to benchmark resolution step.

`-plot` to plot history in iiplot (xxx this need to be generalised, call to gvdr_utils plot xxx).

Following example optimize only solving:

```
ki=rand(20);
RO=struct('nCompt',100,... number of computation for result averaging
```

```
                'maxDomain',2.^[4:7],... parameter 1
                'maxZeros',logspace(-3,1,5),... parameter 2
                'refineStep',3); % refine results to most relevant parameters
spfmex_utils('ofactoptim solve-refine',ki,RO,ofact(1,'lu')); % method,sol
```

extrotor

Purpose External interface to http://www.sdtools.com/help/mpc.html

Bibliography

[1] C. Desceliers, *Dynamique non linéaire en déplacements finis des structures tridimensionnelles viscoélastiques en rotation.* PhD thesis, École Centrale de Paris, 2001.

[2] M. Géradin and D. Rixen, *Mechanical Vibrations. Theory and Application to Structural Dynamics.* John Wiley & Wiley and Sons, 1994, also in French, Masson, Paris, 1993.

[3] J. Batoz and G. Dhatt, *Modélisation des Structures par Éléments Finis.* Hermès, Paris, 1990.

[4] J.-P. Laîné, *Dynamique des rotors.* Cours de l'École Centrale de Lyon, 2005.

[5] A. Sternchüss, *Multi-level parametric reduced models of rotating bladed disk assemblies.* PhD thesis, Ecole Centrale de Paris, 2009.

[6] M. Lalanne and G. Ferraris, *Rotordynamics prediction in Engineering.* Wiley, 1998.

[7] G. Lallement, C. Berriet R., and S., "Updating finite element models using static deformations," *International Modal Analysis Conference*, pp. 494–499, 1992.

[8] R. G. and V. C., "Calcul modal par sous-structuration classique et cyclique," *Code_Aster, Version 5.0, R4.06.02-B*, pp. 1–34, 1998.

[9] Sternchüss, A. and Balmes, E. and Jean, P. and Lombard, JP., "Reduction of Multistage disk models : application to an industrial rotor," in *012502*, 2008. Paper Number GT2008-012502.

[10] E. Balmes, "Orthogonal maximum sequence sensor placements algorithms for modal tests, expansion and visibility.," *IMAC*, January 2005.

[11] A. Sternchüss and E. Balmes, "On the reduction of quasi-cyclic disks with variable rotation speeds," in *Proceedings of the International Conference on Advanced Acoustics and Vibration Engineering (ISMA)*, pp. 3925–3939, 2006.

[12] G. Vermot Des Roches, *Frequency and time simulation of squeal instabilities. Application to the design of industrial automotive brakes.* PhD thesis, Ecole Centrale Paris, CIFRE SDTools, 2010.